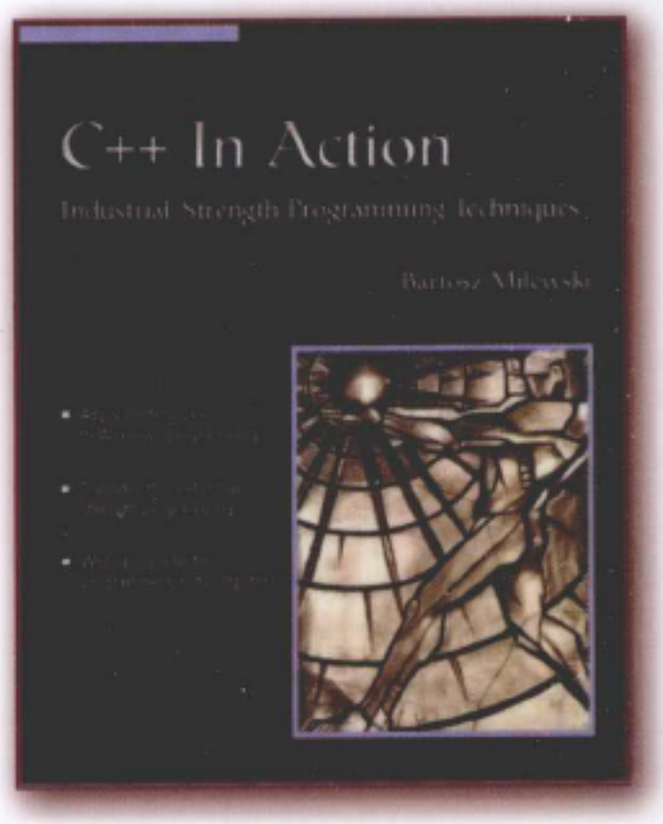
PEARSON

C和C++实务精选
品味岁月积淀，读享技术菁华

C++ 实践之路

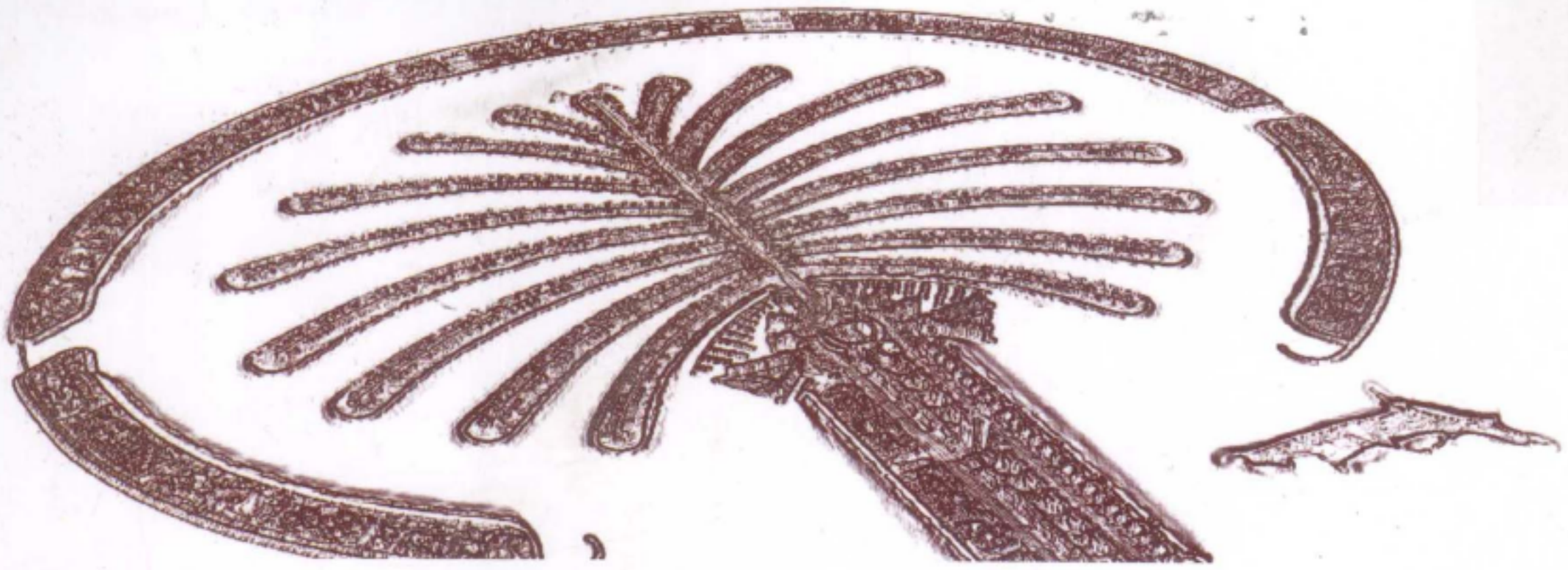
[美] Bartosz Milewski 著　　周良忠 译

C++ In Action:
Industrial Strength Programming Techniques



- 学习工业强度的C++编程技术
- 使用C++设计新思维编写Windows程序
- 学会编写清晰整洁、容易理解的C++程序

人民邮电出版社
POSTS & TELECOM PRESS



本书涵盖了常规C++标准教程没有涉及的知识，为C++编程新手和老练的程序员提供了一个全新的视角来审视C++。本书重点关注的是C++编程技术，它抓住了C++编程的本质，提供了一种用来交付健壮、可靠、优雅的代码的方法论。

本书以项目为基础，首先对C++语言进行了概述，然后描述了如何避免C++编程中的陷阱，以及如何理解、修改和调试C++代码。本书讲解了如何在一个合理的时间表内编写出小巧、紧凑、可靠且可扩展的程序。此外，本书还讲解了使用C++进行Windows编程的大量知识，以及如何使用C++语言来开发大型项目。

作为一本独特而全面的C++编程指南，作者在本书中提供了一些可靠的C++编程技术，其中包括：

- 如何高效使用C++的标准模板库；
- C++中的实现策略，包括自顶向下的面向对象实现；
- 对具有C和Java编程背景的程序员来说，C++中存在的优点和陷阱；
- C++中的资源管理和异常处理；
- 使用C++进行Windows编程，包括将Windows API封装到C++类、命名空间和模板中；
- 如何重构程序，以便可以将其从Windows平台移植到其他平台（如Linux）；
- 大型软件项目的动态管理（从概念到交付）；
- 如何协调团队成员之间的开发。

Bartosz Milewski是Reliable Software公司的总裁，该公司以向程序开发人员提供可以提升生产力的高品质工具为宗旨。在过去几年，他的作品被广泛刊登在主流的期刊上。他在微软公司工作的8年里，他是Windows 2000中Content Index组件的开发负责人。他还曾经在波兰的弗罗茨瓦夫大学讲授C++编程课程，并持有弗罗茨瓦夫大学理论物理学的博士学位。



本书配套光盘包含了书中所有的源代码以及Code Co-op无服务器版本控制系统，它可以运行本书的所有项目。

PEARSON
www.pearson.com

人民邮电出版社-信息技术分社
http://weibo.com/ptpitbooks

分类建议：计算机／程序设计／C++
人民邮电出版社网址：www.ptpress.com.cn

封面设计：任文杰

ISBN 978-7-115-29134-9

ISBN 978-7-115-29134-9
定价：79.00 元（附光盘）

PEARSON

C++ 实践之路

[美] Bartosz Milewski 著　　周良忠 译

人民邮电出版社
北京

图书在版编目（CIP）数据

C++实践之路 / （美）迈尔威斯基（Milewski,B.）著
；周良忠译. -- 北京 ：人民邮电出版社，2012.11
ISBN 978-7-115-29134-9

Ⅰ. ①C… Ⅱ. ①迈… ②周… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第189529号

版权声明

Bartosz Milewski: C++ In Action: Industrial-Strength Programming Techniques
Copyright © 2001 by Pearson Education, Inc.
ISBN: 9780201699487
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior consent of Addison Wesley.
Published by arrangement with Addison Wesley Longman, Pearson Education, Inc.

版权所有。未经出版者书面许可，对本书任何部分不得以任何方式或任何手段复制和传播。
人民邮电出版社经 Pearson Education, Inc.授权出版。版权所有，侵权必究。

C++实践之路

◆ 著　　[美] Bartosz　Milewski
　 译　　周良忠
　 责任编辑　傅道坤
◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
　 邮编　100061　　电子邮件　315@ptpress.com.cn
　 网址　http://www.ptpress.com.cn
　 北京艺辉印刷有限公司印刷
◆ 开本：800×1000　1/16
　 印张：27
　 字数：598 千字　　2012 年 11 月第 1 版
　 印数：1－3 000 册　　2012 年 11 月北京第 1 次印刷
著作权合同登记号　图字：01-2011-7818 号
ISBN 978-7-115-29134-9

定价：79.00 元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

本书将带你领略 C++作为工业编程语言的强大威力。

全书分四个部分，共 23 章。第一部分（第 1 章～第 5 章）从面向对象的角度讲解了 C++的相关语言知识。第二部分（第 6 章～第 13 章）介绍了许多实用的工业强度的编程技术，如清理、隐藏实现细节、资源管理、重载运算符等技术。第三部分（第 14 章～第 18 章）探讨了编写和维护 Windows 应用程序的相关知识，是本书的特色部分。第四部分（第 19 章～第 23 章）在以前所学知识的基础上进一步对软件设计策略、团队协作开发、平台移植等内容进行简单介绍。

附录 A 提供了第一部分内容中部分练习的答案。附录 B 介绍了事务处理相关知识。

本书适合于不同层次的 C++程序员阅读，无论是初学者还是高级程序员，都可从中汲取有用的 C++知识营养。

Preface to the Chinese Edition

I was lucky to have a lot of feedback from my readers since this book was published in the United States. As my readers were trying to learn programming,I was learning about the strengths and weaknesses of my book. Here are some of the things I have learned.

The first part of the book, the introduction to C++, seems to be too difficult for most beginners. If you are a beginning programmer, don't get discouraged the first time your read it. I did use this material when teaching college students. I also used this book to teach C++ to a nephew of mine, who is a high school student, but the truth is I had to do a lot more explaining than the book does. A good teacher may use my book as a basis for a C++ course for the beginners, adding in-depth explanations and developing additional examples.

The traditional way of teaching C++, starting with the C subset, is now widely considered ineffective. My approach was to go straight into object-oriented programming and introduce procedural and generic programming later. This is why I didn't make much use of the C++ Standard Library in the beginning chapters.

There is another school of thought, which lets the students jump directly into the Standard Library and learn C++ as if it were Basic[1] . This way the students can start doing useful things from day one, but they have to accept a lot of features on faith. They learn to use templates, containers, iterators, and so on, without understanding how they work or how they fit into the language.

The best approach depends on the particular audience and probably falls somewhere in between the two schools—introducing a few staples of the Standard Library from the very beginning but keeping the main thrust of the initial course in the area of object-oriented programming. If I were to rewrite my book now, I would probably introduce standard strings in the introductory chapters. Strings and input/output streams are easy enough abstractions—they can be viewed as elementary building blocks of C++ programs.

Advanced programmers gave me a lot of positive feedback on the second part of the book—the Techniques. In particular, the Resource Management techniques described

[1]This modern approach was introduced in *Accelerated C++: Practical Programming by Example* by Andrew Koenig and Barbara Moo, Addison-Wesley 2002.

there turned out to be an eye opener for many. These are simple techniques that make an enormous difference to the quality of programs. I am becoming more and more convinced that Resource Management in C++ is more powerful than garbage collection in other languages.

The chapters on Windows programming brought the most interesting feedback. I've learned that a lot of programmers shy away from Windows programming because they believe that Windows applications can only be developed using one of the commercially available libraries. The quality of such libraries is often very poor, and, as it's often true with programs, whatever they lack in quality they make up for in complexity. Many readers of my book were relieved to see how Windows API can be encapsulated in uncomplicated modern C++ constructs that are both easy to understand and safe to use.

Finally, the last few chapters of the book, which talk about the complexity of software development, are getting more validation as time goes by. Programmers' time is getting progressively more expensive, making the search for better programming methodologies and tools even more urgent.

I am happy to see my book made available to Chinese programmers. I hope it will contribute to the quality and reliability of the software they create.

In my mind, the unique challenge for the Chinese programmer is to make the computer work seamlessly with the Chinese language and, in particular, with the Chinese script. I realize that my book falls short in this area. It doesn't cover such important topics as wide character strings and streams, Unicode, or localization. There are whole areas of C++ (and Windows) to support a wide variety of languages, scripts, and customs. I can only hope that my Chinese readers will supplement the material from this book with additional studies of these topics.

Bartosz Milewski

September 2002

中文版序

自从本书在美国出版后，我有幸收到了许多读者的反馈。读者在利用本书学习编程的同时，我也意识到了本书的优势与不足之处。下面将我的体会奉献给大家。

本书的第一部分“语言”对于大部分初学者来说似乎太难。如果你是一位入门级的程序员，开始阅读这部分内容时不要被吓倒。我在给大学生讲课时使用的就是这部分材料。我还使用本书来教授我的侄子，他只是一名高中生，但实际上，在书本内容之外，我还得进行额外的解释。有经验的教师可以使用本书作为 C++初学者的基础课程，再辅之与相关解释和示例，完全可以达到理想的教学效果。

讲授 C++的传统途径是从 C 的子集开始，现在人们普遍认为这种方式效率不高。我的教学方法是直接切入面向对象编程，并随后介绍过程化和泛型编程。这就是我为什么没有在开始章节中使用太多 C++标准库的原因。

还有另一种教学法，即让学生直接跳到标准库并学习 C++，把这部分内容当成基础知识[1]。按照这种教学法，学生第一天起就可以做一些有用的事，但他们必须首先无条件掌握许多知识点。他们必须学会使用模板、容器、迭代器等，而不一定理解它们的工作方式或如何适应于该语言。

最佳教学方式根据特定的读者而有所不同，也许应该是上述两种方法的折衷——从一开始就介绍一些标准库的主要知识，但重点放在面向对象编程上。如果让我现在重写本书，我大概会在入门章节中介绍标准字符串。字符串和输入/输出流是比较容易的抽象概念——它们可以看作 C++程序的主要构建块。

高级程序员在反馈中对第二部分内容（技术）给予了肯定。特别是，这里描述的“资源管理技术”让许多程序员恍然大悟。这些技术虽然简单，但应用的好坏对程序质量产生的影

[1] Accelerated C++: Practical Programming by Example(Addison-Wesley 2002)一书采用的就是这种现代的方法，作者是 Andrew Koenig 和 Barbara Moo。

响却是天壤之别。我越来越相信，C++中的资源管理比其他语言中的垃圾回收更强大。

“Windows”一部分提到的反馈最有趣。我了解到，许多程序员对于 Windows 编程退避三舍，因为他们认为只能使用商业库才能开发出 Windows 应用程序。这类库的质量往往非常低劣，而且事实表明，质量的低下往往带来程序的复杂性。许多读者在本书中看到 WindowsAPI 可以封装到并不复杂的现代 C++结构中，而且这种作法易于理解，使用安全：这使他们甚感欣慰。

最后谈谈本书的结尾几章。它们讲解了软件开发的复杂性，随着时间的延续，这种复杂性将得到进一步的证实。程序员的时间日益宝贵，找到更好的编程方法和工具甚至显得更为紧迫。

我很高兴看到本书能与中国程序员见面。我希望本书能为他们创建高质量、高可靠性软件出一份力。

在我的记忆中，中国程序员面临的独特挑战是让计算机能与中文，尤其是与中文字符无缝工作。我意识到，对于中国读者来说，本书缺少了这方面的知识。本书尚未涉及到宽位字符串和流、Unicode 或本地化这方面的重要话题。要支持各种不同的语言、字符和习俗，这涉及到 C++(和 Windows)的各个方面。对于这些主题，我只能希望中国读者在看过本书之后，额外地学习其他补充材料。

Bartosz Milewski

2002 年 9 月

序

为什么写这本书

的确，现在详细讲解C++各方面知识的优秀书籍为数不少，为什么还要写这本C++书呢？对于学习这一语言及各种编程技巧来说，市场上提供的书籍已相当饱和了。但本书不是一本语言参考书或高明技巧和模式的集合，本书旨在讨论编程方面的知识。

讲授编程与教授一种语言大不相同。程序员通常要面对一个必须通过编写程序来解决的问题——而不是面对他需要讲解的语言功能。在本书中，我将尽可能演示如何使用C++这一语言作为一种解决编程问题的工具。

本书以几个软件项目为中心展开。在每个项目中，我首先描述即将解决的问题。然后，我讨论程序应该做什么、应该是什么样，以及如何对用户的输入做出反应。在此基础上，我构建一个没有实现具体功能的程序框架。最后，我将按组件逐个实现相关功能。

不过，程序也不会到此停止不前。接下来，本书还要进行一系列的代码审查并进行改写。“怎样才能精益求精”？这是程序员经常自问的问题。再接着另一个需要解决的问题是“如何编写容易修改的代码”？

与现实生活一样，编程的途径也不是唯一的。这就是我们讨论不同解决方案优缺点的极其重要的原因。不知道如何比较不同方案的程序员将举步维艰。当我讨论“因为人们总是这样做……”这样的问题时，我会说出自己遇到的挫折。在本书中，这样的讨论有很多。我努力探讨每种解决方案的优缺点，而且在多数情况下，我设法找到我认为理想的方案。不过，我绝不会以“理想”作为评判目标。我相信，每个人都可以在这些所谓“理想”的背后提出一些非常实际的意见。在多数情况下，一个理想的解决方案进行了非常优秀的抽象或通用化。这样生成的代码易于理解、修改和调试。

最后，在本书中，我强调了编程中人的因素。我的座右铭是“程序是为程序员而写的，不是为计算机而写的。”程序员希望编写更优秀的程序，不是为了让它更易于计算机理解，而

是让程序更易于让人理解。不理解程序是不可能维护程序的。这一点看起来似乎明显，但许多程序员却忽视了这一不言而喻的真理。

为什么适合你阅读

你是哪一类读者？你可能是一个接触过编程但想学习 C++的初学者，可能是一个想要完成大学教育的学生，可能是一个想从学术型上升到工业环境的熟练程序员，或者是一个需要获取新创意的经验丰富的程序员。无论你属于哪一类，我都希望本书能成为你的至爱。

我为什么写这本书

为什么偏偏由我来撰写这本 C++编程书？可能成为本书读者的你有权知道我的资历，尤其因为我不是一个教育界的计算机科学家。

我获得了一个理论物理学博士学位——我的论文是关于超对称非线性 δ 模型。我过去擅长计算超对称费曼图。然后，大约在 1986 年，我开始迷恋计算机，于是我放弃了博士后学位并开始编程。

物理学家是一种介于数学家和工程师之间的人。一个物理学家习惯于穿梭在理论和实践之间。物理学家不应该相信理论，除非他或她在实践中证实了自己的理论，而且他或她总是尽量总结各方面经验来发现一条新原理。尤其重要的是，一个物理学家不应该不加批判地接受一切。

与数学家不同，物理学家必须更现实，更关心过程而不是形式上的理论。物理上的过程就是进行计算。而在编程中，过程就是软件维护。因为我是一个物理学家，所以在本书中，我的讲解重点是程序的改写而不是程序的创建就不足为奇了。

我从 1994 年开始讲授 C++并编写本书，当时我仍然在微软公司工作。我时常造访我的母校——波兰 Wroclaw 大学。每次造访都会促使我撰写更多的章节，我真想在那儿汲取更多的营养。在此期间，我开始将本书转换成 HTML 文档并在网上发布。因为 C++还在发展，我必须一而再、再而三地改写许多章节。例如，当 C++标准库中添加了 STL 时，我必须回过头来重新构思本书大部分代码的实现过程。

当我在微软从事设计并主持索引服务器的开发时，本书所描述的许多技术已经得到了发展。此后，我的新公司（Reliable Software）成为了测试新的编程方法的极好的实验室。本书所描述的所有技术都已集成到我们的产品中。事实上，当每一个更好的新想法出现时，我们都会改写我们的代码。

正因为是一个物理学家，所以我既是一个称职的程序员，同时也是一个不称职的程序员。之所以称职，因为我不断修改我所使用的技术、发现新的通用化方法并彻底探讨编程的各个方面；之所以不称职，是因为当我发现一个更好的解决方案时，我就想改写所有代码（我将尽量让你相信，这一想法实际上并不太坏）。我以前的工作主管总是挑我的刺，因为我质疑每个决定、惯例及习惯。我希望这一同样品质也能让我成为一个优秀的作者。

如果读者对本书提供的代码有更好的解决方案，请与我联系，我可以对本书进行第 n 次的重写。我的 E-mail 地址是：bartosz@relisoft.com。

致谢

真诚感谢 Wroclaw 大学和 Digipen 工学院的我的学生，因为讲授一种语言是学习它的最好途径。感谢 Jerzy Lukierski 教授，是他邀请我回到学校讲授计算机科学的。我同时感谢微软和 Reliable Software 公司的同事，是他们第一次尝试并改进了本书中所描述的技术。感谢我的妻子 Pam，除了不遗余力地支持我以外，她还帮助我编辑了本书的第 1 章，使这一章更加简单易懂。最后还要感谢 Addison-Wesley 出版社的编辑，尤其要感谢的是 Debbie Lafferty。

译者序

软件项目的成功与否与开发语言的选择息息相关。而在确定某种语言为开发工具前，我们通常要提出这样的问题“这一语言能高效地实现我们的软件目的吗？最终产品是否易于维护？它与其他语言相比优势何在？”当今的很多商业软件首选 C++作为他们的开发语言，就是因为 C++对以上问题能提供令人相当满意的答案。简言之，C++是 ANSI 标准 C 语言的增强版本，但它对 C 语言的扩充是革命性的。C++不仅支持面向对象编程，而且它还可以方便地构建一个由相关对象组成的层次等级树，同时，它出色的可维护性和可扩展性使得它成为商业软件开发中最流行的语言之一。

要想成为一名优秀的 C++程序员，仅仅具备 C 的相关知识背景是远远不够的，而且 C 的一些习惯思维方法并不利于充分利用 C++的高效与先进性。因此，学习一些 C++编程技巧是每个合格 C++程序员的必修课。本书正是一本为我们提供大量实用 C++编程技巧的书籍。

本书第一部分从面向对象的角度讲解了 C++的许多语言要点，初学者可以对 C++面向对象编程获得更清晰的认识。第二部分则讲解了大量实用技术，这些技术经过广大程序员的多年实践证明是开发可靠程序的必备技巧。例如，资源管理技术是任何用 C++开发的商业程序必须正确处理和应用的核心技术之一。本书的另一个可贵之处是对 Windows 应用程序开发的诸多技巧进行了一定深度的介绍。Windows 应用程序的开发难度大，同时，相关资源也难以获取，所以，这部分知识对于专业程序员来说无疑具有宝贵的参考价值。最后，作为一名工业强度型程序员，他还必须具备软件设计、团队协作开发、平台移植等多方面的知识，本书也对这方面的知识进行了讨论。

为了方便读者的学习与实践，本书还附带一张光盘，你可以从光盘上找到本书所涉及的所有源代码及项目。

由于译者水平有限，错误在所难免，望广大读者不吝指正。

前言

我将本书分成 4 个部分：语言、技术、Windows 和知识扩展。下面简要描述每部分的目标和结构。

语言

本书的第一部分重点讲解作为通用目的编程的 C++语言，然而这不是你常见的 C++指导。

对于那些对 C 或 C++知之甚少的初学者来说，本部分内容只是作为一种面向对象的语言介绍了 C++（尽管 C++的内涵远非如此）。本部分没有重点介绍语法或语法规则，它只是说明如何在 C++中表达一定的思想，就像通过对话而不是背单词和语法规则来学习一门外语一样（当我给学生授课时，我称这部分内容为“对话式 C++”）。毕竟这是程序所需要的：能以特定语言编写程序的形式来表达自己的思想。当我学习一门外语时，我想知道的第一件事就是如何使用这样的问句“How much does it cost？”我不需要学习“cost”的过去时、现在时及将来时的所有动词时态，我只希望能在国外顺利地购物。

对于一个不太了解 C++的 C 程序员来说（不等于说 C++不活跃、很神秘，但 C 的其他一些子集往往有这种弊病），这是一个忘记 C、有效运用 C++编程的练习机会。为什么要忘记 C？C++不是 C 的一个超集吗？的确 C++是 C 的一个超集，让 C++与 C 兼容完全是一个符合实际市场决策，而且这样做取得了成功!它不是一个全新的产品，它不必花费十年的时间来单独开辟市场，它是“3.1 版本”的 C。这样做也是利弊相随的。“利”在于 C++及一些面向对象的编程元素与 C 向后兼容，这样可以迅速在编程领域占领一席之地“弊”在于它并不需要程序员更改编程习惯。

不必一次性重写所有的现有代码，许多公司可以逐步过渡到 C++，现在很多公司仍然如此。通常的过渡方式就是将 C++当成一种“更严格”的 C 来引入。从原则上讲，所有的 C 代码都可以作为 C++代码重新编译。实际上，C++有着更严格的类型检查，而且编译器可以检测到更多的错误、发出更多的警告。所以，使用 C++编译器重编译 C 代码是一个清理现有代

码的好办法。在此阶段对源代码要做的修改主要是更正错误、实施更严格的类型。如果代码是用前 ANSI C 写成的，则会生成所有函数的原型。在这一 ANSI 化的过程中发现的错误之多让人感到吃惊。为了顺利过渡，以上所有工作都是值得的。只有在缺少优秀的 C++编译器时（当然这种情况很少发生）人们才去使用 C 编译器。

一旦过渡到 C++编译器环境中后，程序员迟早要学习一些新技巧，并最终开发一些 C++编程方法（或通过自学、阅读一些辅助性书籍来达到目的）。但此时，许多程序员会受到一些不良的影响。因为 C++的一些子集是许多原来 C 程序员应用过的（我将此子集称为“C 特区”）。许多 C 程序员看到这些“C 特区”后开始憎恨 C++，但他们并没有意识到 C++同样具有很多优秀之处。

在本书的前言中，我想澄清一件事。对于一个“C 特区”程序员来说，本书对他们应该是一种冲击（我希望如此）。从本质上讲，“你到目前为止所做的一切都是错的”、“Kernighan 和 Ritchie 并非神”（Brian Kernighan 和 Dennis Ritchie 是 C 的创建者，也是著名的 *The C Programming Language* 一书的作者）。我知道，这类程序员的第一个自然反应就是合上本书立即要求退款。不要冲动!本书打破习俗的用意不在于伤害某个人的感情；而在于以一种不同的哲学思维来提醒读者重新思考他们的信条。

对于 C++程序员来说，“语言”部分提供了一个审视 C++的新角度。它演示了如何避免 C++的缺陷以及如何根据先设计后具体实现的方式来应用这一语言。如果我说 C++是一种优美的编程语言，这是不真实的。但它即将成为编写正规软件的最流行语言（至少在某些时候是）。我们最好充分利用它的丰富表达力来编写更出色的软件而不是用它不利的一面来伤害我们自己。对于一个 C++程序员来说，这一部分内容不难读懂。而且，虽然这里介绍的结构和技术已广为人知，我还是尽量从不同的角度来展示它们。我的主导思想是创建一个易于维护、可读性强的系统。这就是为什么我抓住每一次机会不仅演示不同的编程选择，而且还解释为什么优先考虑某种方案的原因。

最后，对于一个 Java 程序员来说，他们可以利用本书来扩展知识。事实表明，根据一定的原则，利用 C++可以编写出安全和可靠的代码。Java 能做到的，C++也能做到，而且能做得更多。另外，C++能提供无可匹敌的操作性能。

然而操作性能不是投身 C++的唯一原因。完善的资源管理在 C++中能实现，但在 Java 中不可能实现，因为 Java 要依赖于垃圾回收机制。在 C++中，你可以创建这样的对象：它的生存期由它所在的作用域准确定义。在退出这些作用域时，你要保证这些对象将被销毁。这就是为什么你可以让 C++对象作为一个信号量来负责处理这些重要资源（如文件句柄、数据库事务处理等等）的原因。Java 对象有一个未定义的生命期——只是在运行时决定回收它们时才释放这些资源。所以，说到 Java 中处理资源的方式就得重提旧式 C 的异常范例，在这种情况下，最后的子句必须要进行痛苦的显式垃圾回收。

没有“纯粹”C++出身的程序员。当我们“谈到”C++时，或多或少都会涉及其他一些编程背景。我们中间的一些人有着深厚的 C 背景，一些人使用类似 Smalltalk 的表达方式，还

有一些人在使用 LISP。“语言”部分的目标是尽可能让你成为一名正统的 C++程序员。语言是表达思想的一种工具。所以，我的讲解重点不是语法而是程序员表达自己的方式。在这里不是介绍“C++的结构是什么，你应该怎么用”，而更多的是以“我们有了一种思想，那么该怎样用 C++来表达呢？”的方式来讲解。最初，这一“思想”可能以“星球是一种天体”或“一个堆栈允许你用推入或弹出数据”的简单句子形式表达，以后再将这些句子组成“段落”和“章”来描述软件组件的功能。在需要解决相关问题的适当场合介绍不同的 C++结构。

技术

开发优秀软件不仅仅需要了解语言本身。首先，程序不是在真空中执行，它必须与计算机交互作用。交互意味着通过操作系统来发挥作用。若对操作系统一无所知，就不可能编写出像样的程序。其次，我们不仅需要编写能运行的程序，我们还希望程序小巧、快速、可靠、强健以及具有扩展性。最后，我希望在合理的时间内完成程序的开发，并且能在以后维护和改进此程序。

本书第二部分“技术”的目标是让你从“业余编程”过渡到“工业强度型编程”。我描述了一些可以使程序更强健、维护性更强的 C++编程技术。其中的“资源管理”重在介绍程序的创建、获取、拥有及释放各种资源。在程序执行的任一时刻，每一种资源都必须存在一个明确的拥有者负责释放它。这一简单的思想在设计和维护复杂的软件系统中相当有用。运用资源所有权关系分析法可以避免、发现和排除许多 bug。

资源管理与 C++异常处理能自然地吻合。事实上，如果不封装资源几乎不可能使用异常来编写合理的 C++程序。那么，什么时候使用异常呢？它们能为我们做些什么呢？这取决于你对下面简单问题的回答：你总是检查 new 的结果吗（或者对于 C 程序员来说就是检查 malloc 的结果）？这是一个反问句。除非你是一个格外小心的程序员，你才不用这样!这意味着不管你希望与否，你都已经使用了异常，因为访问一个空指针就会导致一个叫做“一般性保护”（程序员也称做 GP 过错或访问违例）的异常。如果你的程序没有添加防异常措施，当它遇到这种异常时就会死机。甚至操作系统会弹出一个消息框，证实你的应用程序所使用的技术不合格。

我的观点是，为了编写强健可靠的应用程序，使用异常是最终的选择——这也是本书的目的。当然，还有其他一些可成功地应用于开发强健可靠应用程序的编程技术（这些技术现在仍然被我们使用）。不过，结合应用 C++异常与资源管理带来的简单性和可维护性是这些技术所不及的。

Windows

就我所知，这是第一本深入探讨了使用现代 C++进行 Windows 编程的书籍。事实是 Windows API 很混乱，现有的库也同样不是完全规范一致，使得 Windows 应用程序的编写不是那么简单，但这并不意味着这一主题就没有意义。在 C++类、名字空间及模板中封装

Windows API 是一种挑战，这需要对整个 Windows 范例的重新深层构思。

这部分内容的主要目标是探讨创建和维护 Windows 程序的简单途径——更侧重于维护。我常常问我自己这样的问题："添加或修改一个消息句柄应该简单到什么程度？添加一个新菜单项、一条命令、一个对话框呢？而且如何才能将程序员犯错的可能性降到最低呢？"最后，还要考虑将库移植到例如像 Linux 的其他平台上（但这不是必需的）。

本书的 Windows 部分反映了一个不断改进的工作。事实上，这一工作多少年来一直在不断改进。对本部分所描述的每一个思想都有若干种不同的实现方法，并进行了测试（通常集成在某一产品中），也可能由于找到了更好的方案而最终放弃了。

知识扩展

在这一部分内容中，我涉及了软件开发的许多扩展知识。我侧重讨论软件项目的动态性，既从管理和规划的角度入手，同时着眼于开发策略和技巧。我从软件项目的概念到发行描述了其动态性。我讨论了文档、设计过程以及开发过程。不过，我不想提供一个现存的方案，因为这样的方案不会在任何时候都适用——原因如下所述。

成功开发一个应用程序（或系统）不是光靠学习一种语言、掌握其一定的技术就足够了。当今的商业软件项目是人类从事的最为复杂的工程之一。编程实质上是一门处理复杂性的艺术。很多人尝试使用传统的工程方法来控制软件的复杂性，模块化、软件重用、软件 IC（integrated circuit，集成电路）等——但我们要面对这样的事实：通常它们不能正常发挥作用。它们可能对提供低层构建块和库非常有用，但它们很难用作设计和实现复杂软件项目的指导原则。

原因很简单：软件块中的重复很少。试着比较一段程序的打印输出与一幅微处理器晶片图，你可以发现微处理器布局上的许多重复模式。这些重复块仿佛某种高科技晶体。另一方面，精简后的程序看起来更像一个高科技分形结构。你可以见到许多相似性——大模式与小模式相似。但你很少发现有完全准确的匹配或完全相同的重复。每个小块都像一个单独的手工艺品。程序中的重复不仅没有必要，而且还会给今后的维护带来困难。如果你修改某段代码或排除某个错误，你就要审查这一段代码的所有拷贝，而且还要同时对这些地方做出修改。

重复的弊病还反映在软件的生产过程中。软件工业中的研究、设计和制造不同于其他工业。例如制造的作用并非主导。严格来说，电子化发行渠道可以使制造阶段与之完全无关。R&D（研究与开发）扮演着重要角色，这一点比许多其他工业更突出。但真正区别软件开发与其他工业的是在产品中体现的设计成分。编程就是设计。设计、构建原型，一遍遍地测试。软件工业是最终的"设计工业"。

程序员在人们心中往往是这样一种灰色形象：独自夜战、封闭自我而且生活邋遢。我认识这样的程序员，但我仍然相信还有另一类程序员。这种旧文化的残留现在逐渐灭绝了，因为硬件的进步、软件的发展使得孤军奋战开发出有用的可靠程序已不可能。团队工作是软件开发必不可少的。

分解整个工作并在团队工作中协作努力总是一个大的挑战。在传统工业中，团队中的成员知道（至少在理论上）他们要做什么。他们知道例行程序。他们听着音乐，知道下面的步伐，并跳出同步的舞蹈。在软件工业中，团队成员即兴跳出舞步，同时为其余的队友谱出相应的“音乐”。

我提倡改变软件开发的重心。我将尽量让你信服这样的思想“程序是为程序员而写的”，而不应该遵循陈旧的“程序是为计算机而写的”的思想。这一论点是整本书的基点。如果你不把你编写的代码看作让其他程序员阅读、理解和修改的出版物，那么你就不可能开发出工业强度型软件。没有人希望自己编写的代码成为一段密码。计算机可以编译和运行你的程序，还要求你的同事可以理解它的意义。而且至关重要的是，只需极少的努力就可完成这一点，因为不理解代码就不可能开发和维护软件。

如何使用本书

缺乏足够的经验不可能学好编程。这是为什么本书附带光盘中包含所有源代码及相关软件的原因，它们可以供读者进行练习。如果你安装了我推荐的编译器之一，你就可以构建和运行本书涉及的所有程序。尤其是开始的一些例子以一套文件的形式提供给大家，它们保存在各自的目录中。你可以从光盘将这些文件复制到本地硬盘中并编译它们。不过，大部分例子需要一个版本控制系统。

版本控制系统

本书中的大部分代码示例属于一个软件项目，随着讲解的深入，这些代码得到了一步步的改进。提供这些逐级改进版本的最佳途径就是使用一个版本控制系统（Version Control System，VCS）。VCS 是软件开发的基本工具之一。它可以让你跟踪所有的变化、恢复到旧版本，而且更重要的是，可以与其他程序员相互协作。

随书光盘上赠送了 Reliable Software 公司开发的一种 VCS，这就是 Code Co-op。因为 VCS 是一个商业产品，所以通常你要购买使用许可证才能使用。不过，把它作为一个项目的“观察器”使用就不必获得使用许可证，而且这已足以让你学习本书中的项目了。

Code Co-op 有一些独特的功能，因此尤其适合于本书的学习。在安装 Code Co-op 的过程中，将你的计算机作为一个单机来配置。接下来，你就可以在学习软件项目过程中与我“交互协作”了。这种协作是单方面的：我开发了此项目，你将得到了同步脚本复制，它们描述了代码所作的修改。

注意，安装 Code Co-op 不会覆盖计算机系统中的任何 DLL 文件。运行卸载程序后，Code Co-op 将从你的计算机上彻底消失。

编译器

在本书成稿之时，集成了性能良好的 IDE（Interactive Development Environment，交互式

开发环境）的标准兼容 C++编译器并不多。我已经用 Metrowerks CodeWarrior 6.0 和 Microsoft Visual C++ 7.0 beta1 测试过了所有的示例代码（这些代码不能保证与早期的 VC++版本兼容）。

我为两种编译器提供了相关的项目文件。如果你使用的是 CodeWarrior，则使用扩展名为.mcp 的文件作为项目文件。如果使用的是 VC++ 7.0，则使用扩展名为.vcproj 的文件代替。

安装以上任何一种编译器之后，双击项目文件就可以打开它。然后可以浏览源文件和构建项目（使用“Build”按钮）。构建项目后，你可以运行得到的可执行文件（使用“Run”按钮）。两种 IDE 都有内置调试器，它们不仅有助于调试程序，而且可用于（单步）跟踪代码。

代码示例

world1

当可以在光盘上找到本书所讨论编程例子的源代码时，本书就会以边注（一个光盘图标加上对应的目录名）的形式指出它们在光盘上的对应位置。例如，当你看到边注“world1”时，你就可以查找光盘上 src 目录下的子目录 world1，得到相应的源代码（在光盘的启动界面上单击“Source code”链接可进入 src 目录）。将此目录下的内容复制到本地硬盘上，然后可以构建并运行它。

Project Calculator

不断开发改进的示例代码以 Code Co-op 项目的形式提供。当你看到一个指定了项目名的边注时就可以启动一个新项目。例如，当你看到边注“Project Calculator”时，就可以放入光盘，单击启动画面上的“Create a Project form the book”，接着出现一个项目列表，从列表中选择项目“Calculator”之后，你就进入了此项目，Code Co-op 会显示项目的初始状态。在“文件区”（Files Area）双击文件可以查看它的内容。

Script 1

在 Code Co-op 的“邮箱区”（Mailbox Area）你可以看到许多与不断增加的代码修改对应的同步脚本。每个脚本都有一个编号。当看到一个指定了脚本编号的边注时，你就可以在光盘上打开对应的脚本。例如，当你看到一个边注“Script 1”时，你就应该正在学习项目“Calculator”，你可以进入 Code Co-op 的“邮箱区”打开标题为“1. Implemented Scanner”的脚本链接（此脚本将被标记为“Next”。打开“script 1”后，“script 2”也被标记为“Next”表示下一个可打开对象，依此类推）。按下倒置的信封图标按钮可以完成打开。

打开某个脚本后，你可以直观地看到对此项目所作的修改。在 Code Co-op 的“同步区”

（Synch Area）可以看到一个更改文件列表。当你双击这样一个文件时，你将看到加亮了修改处的不同视图。当你审查完代码后，通过单击“同步”区的“accept changes in all files”按钮接受这一同步修改。

如果你想体验某个版本的项目，你可以将这一项目版本的文件导出到硬盘上的一个单独目录中再来修改它们。进入 Code Co-op 的“历史区”（History Area），选择你想导出的版本（它可能是当前版本）并从“Selection”菜单中选择“Export Version”。

从 Code Co-op 的“Help”菜单中可获得更详细的帮助信息。

排版约定

本书中的所有代码示例均以等宽字体印刷，如：

```
int main(){}
```

代码示例中的一些元素以黑体来强调：

```
World smallWorld;
```

这些需要强调的元素通常是在上下文中进行讨论的新添加元素。

类、变量或函数的名字（如 World、smallWorld、main）在正文中以 Times New Roman 字体印刷。文件名也使用了同样的字体，如 world.cpp。

目录

第一部分 语 言

第 1 章 对象和作用域 …… 3
1.1 全局作用域 …… 3
1.2 本地作用域 …… 8
1.3 嵌入对象 …… 14
1.4 继承（Inheritance） …… 16
1.5 成员函数和接口 …… 18
1.6 成员函数作用域 …… 20
1.7 类型 …… 26
1.7.1 小结 …… 27
1.7.2 练习 …… 27
1.8 抽象数据类型 …… 29
练习 …… 33
第 2 章 数组和引用 …… 34
2.1 引用 …… 34
2.2 运算符 …… 39
2.2.1 算术 …… 40
2.2.2 逻辑 …… 41
2.2.3 位逻辑运算符 …… 42
2.3 基于堆栈的计算器 …… 43
2.3.1 外部规范 …… 43
2.3.2 设计 …… 43
2.3.3 简短的实现 …… 44
2.3.4 实现 …… 47
2.3.5 练习 …… 52
第 3 章 指针 …… 54
3.1 指针的不利之处 …… 54
3.2 指针与引用 …… 55
3.3 指针与数组 …… 56
3.3.1 谈谈汇编 …… 59
3.3.2 练习 …… 60
3.4 指针与动态内存分配 …… 61
3.5 动态数据结构 …… 64
3.5.1 动态堆栈 …… 64
3.5.2 链表 …… 67
3.5.3 字符串表 …… 73
3.5.4 字符串缓冲器 …… 75
3.5.5 表查找 …… 77
3.5.6 哈希表 …… 78
3.5.7 测试程序 …… 80
3.5.8 练习 …… 80

第4章 多态 82
4.1 is-a 的意义 82
4.1.1 谈谈实现 83
4.1.2 开销 84
4.2 解析树 85
4.2.1 谈谈 C 88
4.2.2 练习 91
第5章 一个小型软件项目 93
5.1 开始一个软件项目 93
5.1.1 设计规范 93
5.1.2 简短实现代码 95
5.1.3 扩展程序片断 97
5.2 扫描器 101
5.3 符号表 106
5.4 存储 108
5.5 函数表 111
5.6 节点 117
5.7 解析器 119
5.8 主函数（main） 126
5.9 聚集初始化 127
练习 128
5.10 过程化编程 130
5.11 运算符重载 131
5.12 按值传递 132
5.13 值语义 136

第二部分 技 术

第6章 清理 145
6.1 输出解耦 145
6.2 与“防错性编程”作斗争 148
6.3 完备性编程的一个实例 150
6.4 补充 154
6.5 改进类之间的通信 155
6.6 更正设计缺陷 159
第7章 隐藏实现细节 165
7.1 使用嵌入类 165
7.2 组合类 167
7.3 使用名字空间组合 168
7.4 在枚举中隐藏常量 170
7.5 在本地变量中隐藏常量 172
第8章 共享 173
8.1 分离全局程序参数 173
8.2 增强包容性 173
8.3 模板 176
练习 180
第9章 去掉限制 181
9.1 动态数组 181
9.1.1 动态 MultiNode 185
9.1.2 动态 StringBuffer 186
9.1.3 动态 SymbolTable 188
9.1.4 动态 Store 189
9.2 标准矢量 190
第10章 资源管理 195
10.1 异常 196
10.2 堆栈展开 198
10.3 资源 202
10.4 资源所有权 203
10.5 访问资源 204
10.6 智能指针 204
10.7 所有权转移：第一次尝试 206
10.8 所有权转移：第二次尝试 207
10.9 安全容器 213
10.10 迭代器 217
10.11 错误传播 219
10.12 转换到资源管理 221

10.13 结论 ····· 222
第 11 章 使用标准模板库 ····· 223
11.1 引用计数以及即写即复制 ····· 227
11.2 结束限制 ····· 233
11.3 去掉 C-字符串 ····· 234
11.4 探讨流 ····· 236
第 12 章 持久性 ····· 240
12.1 Calculator 对象 ····· 240
12.2 命令解析器 ····· 243
12.3 串行化与反串行化 ····· 246
12.4 内存中的串行化及反串行化 ··· 256
12.5 多重继承 ····· 257
第 13 章 重载运算符 new ····· 259
13.1 特定类 new ····· 259
13.1.1 缓冲 ····· 259
13.1.2 批量分配 ····· 263
13.1.3 数组 new ····· 266
13.2 全局 new ····· 266
13.3 跟踪内存泄漏 ····· 268
13.3.1 调用输出 ····· 273
13.3.2 方位 new ····· 273
13.4 结论 ····· 274

第三部分 Windows

第 14 章 库 ····· 277
宏和向导 ····· 277
第 15 章 编程范例 ····· 279
15.1 Hello Windows ····· 280
15.2 封装 ····· 286
第 16 章 通过 C++控制 Windows ····· 288
16.1 模型—视图—控制器 ····· 288
16.1.1 控制器 ····· 294
16.1.2 客户代码 ····· 295
16.2 异常定义 ····· 297
16.3 清理 ····· 298
第 17 章 绘图 ····· 300
17.1 应用程序图标 ····· 300
17.2 窗口绘图和视图对象 ····· 304
17.3 画布 ····· 305
17.4 WM_PAINT 消息 ····· 308
17.5 模型 ····· 310
17.6 捕获鼠标 ····· 313
17.7 添加颜色和修饰 ····· 319
第 18 章 一个 Windows 应用程序 ····· 327
18.1 把计算器移植到 Windows 环境下 ····· 327
18.2 用户界面 ····· 328
18.3 子窗口 ····· 329
18.4 Windows 控件 ····· 330
18.4.1 静态文本 ····· 330
18.4.2 编辑控件 ····· 330
18.4.3 窗口子类 ····· 331
18.4.4 插入计算器 ····· 332
18.4.5 列表框控件 ····· 333
18.4.6 模块—视图反馈环 ····· 333
18.5 命令和菜单 ····· 336
18.6 动态菜单 ····· 339
18.7 对话框 ····· 346

第四部分 知识扩展

第 19 章 关于软件 ····· 355
19.1 复杂性 ····· 355
19.2 软件的分形本质 ····· 357
19.3 项目的灵活性 ····· 358

19.4 程序员的灵活性……360
第 20 章 设计策略……362
20.1 自顶向下的面向对象设计……362
20.2 用户界面……363
20.3 需求规范……364
20.4 体系结构规范……365
第 21 章 使用标准模板库……366
21.1 生产效率……366
21.2 团队策略……368
第 22 章 实现策略……371
22.1 确定全局……371
22.2 自顶向下的面向对象实现
方法……373
22.3 程序修改……373
22.4 代码继承……374
22.5 多平台开发……375
第 23 章 移植……376
23.1 创建一个抽象层……376
23.2 移植计算器……377
23.2.1 消除简单的 Windows 依赖性……377
23.2.2 特殊 Windows 依赖性……379
23.3 移植的好处……380

附 录

附录 A 练习答案……383
附录 B 事务处理……400

第一部分

语　言

本书此部分内容是 C++学习指南。它讲解了 C++语言的基础知识，引导读者从学习编写简单例程到开发独立的程序。将符号计算器这一程序作为贯穿全书的一个主要软件项目。

我尽量既让初学者对这部分指南感兴趣，又让有经验的程序员乐于阅读。本指南直接讲解与C++有关的面向对象编程知识和语言特色，包括类、构造函数、引用等。在多数情况下，我不仅要解释如何编程，而且还要告诫不要怎样编程。在讲授像 C++一样的综合性语言过程中，列举一些反面例子很有必要。

第1章 对象和作用域

宇宙中最重要的东西是什么？是物质吗？似乎任何事物都是由物质组成的——银河、恒星、行星、房子、汽车，甚至是我们程序员。但如果没有能量，物质又将变成什么呢？没有能量，宇宙将消亡。能量是变化、运动和生命之源。另外，如果没有空间和时间，物质和能量又将变成什么？我们需要空间来存放物质，需要时间来观察物质的变化。

编程就像创造宇宙一样。我们需要物质：数据结构、对象以及变量。没有作用于对象上的代码，此对象将消亡。所以，我们需要能量，它是程序的生命力，它以可执行代码的形式表现出来。对象需要空间来存放，以便它们相互关联。最后，代码行需要时间来执行。

程序的“空间—时间”用作用域（scope）来描述。对象由它的作用域决定其生存和消亡。可执行代码行在作用域内操作。作用域为程序提供结构。程序最终是由许许多多的结构组成的。

1.1 全局作用域

类定义、*对象定义*、*构造函数*、*析构函数*、*输出流*、*include*、*main*。

在讲授 C 时有一个老传统，这就是回顾 Kernighan 和 Ritchie 的 *The C Programming Language* 一书，编写第一个打印问候语 Hello World!的程序。我们的 C++程序也应该打印这一问候语。当然，我们要创建“the World”，并让程序自己“说出”问候语。

下面的程序实现了以上功能，但同时，它也是 C++编程的一个借喻。每个 C++程序本身就是一个剧本，我们定义了剧本中的角色，并让它们交互作用。从某种意义上讲，此程序是“所有 C++程序之母”。它只包含一个演员，并让它经历创建和删除两个过程。程序在计算机屏幕上通过打印信息与我们交互。当它被创建时，就在屏幕上打印 Hello!；当它消失时，就打印 Goodbye!。现在我们开始编写此程序。让我们先看看下面这个短小程序的代码清单，并逐行阅读对应的解释。

```
#include <iostream>
```

```
class World
{
public:
  World() { std::cout << "Hello!\n"; }
  ~World(){ std::cout << "Goodbye!\n"; }
};
World TheWorld;

int main(){}
```

表 1-1 标识了程序的各个部分。

表 1-1　　识别程序各部分

程序组成部分	程序文本
include 指令	`#include <iostream>`
类定义	`class World` `{` `public:` `  World() { std::cout << "Hello!\n"; }` `  ~World(){ std::cout << "Goodbye!\n"; }` `};`
对象定义	`World TheWorld;`
main 函数	`int main(){}`

让我们从结尾即从 main 开始向前进行讲解。每个 C++程序必须有一个 main 函数，甚至对于这个特殊的程序，即使它没有完成任何实质性的任务，也应该包含一个 main。

你也许对数学中的函数比较了解。函数带有一个参数（或多个参数）并且返回一个值。一个正弦函数以某个角度为参数，返回根据此角度构成一个三角形的两边长度的比值。例如，sin（π/4）返回 0.7071（或者更精确地说，返回 sqrt（2）/2）。同样，在 C++中，函数不带或带有多个参数并返回一个值。参数放在小括号中；如果函数不带参数，则小括号为空。在此例中，main 不带参数。

不同的函数返回不同类型的值。正弦函数返回一个实数，但平方根可能返回一个复数（例如，-1 的平方根为 i，i 为虚数单位）。在逻辑学上，如果 p 为 false，则函数 not（p）返回布尔值 true；如果 p 为 true，则 not（p）返回布尔值 false。在 C++中，函数的定义必须指定它的返回值类型。在后面，我们将讨论不同的类型，但现在，你只要明白 int 是表示整数的内置类型之一就行了。在 C++中，函数 main 必须总是返回 int 类型值。

函数的定义不仅要描述参数及返回值类型，而且还要包含计算此值的命令。这些命令总是用一对大括号{和}括起来（大括号也称为花括号）。对于此例，在 main()后面的大括号中没有任何语句。即不存在任何命令——在此程序中，main 不完成任何任务。通常一个返回那改天 int 值的函数总要给出一些如何计算此 int 值的命令（或者只是简单地将此值设置成任意

数)。特殊函数 main 是一个例外。如果程序员没有定义它的返回值，则将之理解成返回 0（我们将在后面讨论特定返回值的意义)。

C++程序的格式并非语言的要求，而是一种约定俗成。例如，我用最少的空格来编写 main 的定义：

```
int main(){}
```

注意，在 main 和小括号之间、小括号和大括号之间没有任何空格，或者将它分成多行：

```
int
main
(
)
{
}
```

我不选择以上任何一种样式，因为不同的格式对可读性影响很大。在本书中，我尽量遵循 C++程序员广泛采用的格式约定（虽然它们远没有达到通用的程度)。

那么，如果程序中的 main 没有做任何事，我们为什么还需要它？因为执行 C++程序总是经历以下三个阶段：main 前、main 及 main 后。“真正的工作”应该在 main 中完成。但是，main 前和 main 后部分可能代码更多，这一点我们稍后就会看到。

代码行：

```
World TheWorld;
```

定义 TheWorld 是类 World 的一个对象。你也可以说，TheWorld 是类 World 的一个实例。注意，对象的定义后面有一个分号（;)。

对象是面向对象编程中的主要演员，事实上，此行代码是整个程序的中心。如果你给一个 C 程序员演示此程序并询问它的作用，此程序员将说“什么也不做!”。因为 C 程序员查看 main 时，发现里面为空。受过训练的 C++程序员将留意 TheWorld 的全局定义，并说“哈!它将打印‘Hello!’ 和 ‘Goodbye!’”。当我谈到“全局”时，指的是“任何大括号的外部”。大括号外面的所有空间都被看作是全局作用域（global scope)，参见图 1-1。

```
#include <iostream>
class World
{
 public:
   World ()  { std::cout << "Hello!\n"; }
   ~World () { std::cout << "Goodbye!\n"; }
};
World TheWorld;

int main ()  { }
```

图 1-1 全局作用域为大括号外的所有区域（阴影部分）

下面按逆向分析，我们将着重讨论类 World 的定义。类是由程序员定义的一种类型。我们已经看到一种类型：int，这是语言的一种内置类型。World 不是内置类型，所以，程序必须对它进行定义。

将 World 的定义放在大括号中，随后再加上一个分号。关键字 public 的意思是，我们没有隐藏任何东西（关键字是特殊的保留字，它们是 C++语言的一部分）。在后面，我们将了解数据隐藏的重要性，但是现在我们不隐藏任何对象。

类定义中的第一件事就是定义构造函数（constructor）。

```
World(){ std::cout << "Hello!\n"; }
```

构造函数是每次创建此特定类的某个对象时被执行的一段代码。如果给同一个类创建多个对象，将多次调用此构造函数。构造函数看起来像函数，但在它前面没有返回类型。构造函数总是与类同名。构造函数可以带有参数，但在此例中，它不带参数——后面的小括号为空。此构造函数实际上完成了某些任务——因为在后面的大括号中包含有代码，这就是如下命令：

```
std::cout << "Hello!\n";
```

它的作用是将字符串“Hello!\n”发送到对象 std::cout（注意，命令后有一个分号）。预定义对象 std::cout 表示标准输出，假定为计算机屏幕，它将打印所发送的任何内容。一个字面字符串，如“Hello!\n”，它总是放在双引号之中。字符串结尾处的\n（反斜杠 n）意思是“打印一个新行”，所以，后面的打印输出将另起一行。

类定义的下一个是析构函数（destructor），它总是与类同名，但前面带有一个~（一个波浪号）。每次删除特定类的对象时就执行此析构函数。如果要删除同一个类的多个对象，就多次调用此析构函数。尽管析构函数与函数看起来相似（但注意，它的前面没有返回类型），然而，它永远不带任何参数——析构函数名字后面的小括号内总是为空。World 的析构函数完成了某些任务，即打印字符串“Goodbye!\n”。

最后，在文件的顶端有一个 include 指令：

```
#include <iostream>
```

include 指令通知编译器查找名叫 iostream 的文件，并将它包含在这里。严格说来，iostream 之类的“标准”头不必当成一个文件来实现——编译器可以立即生成适当的代码。如果正确地安装了编译器，它将根据编译器系统配置在标准位置之一找到此文件。编译器需要利用此文件来得知 std::cout 是什么以及能做什么。这一对象（类 ciostream 的）是 C++标准库的一部分，并不是语言的一部分。我们将在后面章节看到 std::cout 的更多应用。

我们分析此程序的顺序并不是随意的。它称为自顶向下（top-down）法，被认为是查阅和分析程序（以及编写程序）的正确方式。根据自顶向下的方式，从最一般、最高级别对象开始，再慢慢下降到更具体、更低级别的细节上来。

现在，让我们分析执行此程序过程中发生的一切。

舞台搭建完毕；主要演员是全局对象：TheWorld。它的角色由类 World 定义。类的定义告诉我们构建 World 时它该做什么，当删除它时又该做什么。演出开始了，让我们看看发生

了什么：

- main 前：构建所有的全局对象。在此例中，执行 TheWorld 的构造函数并在屏幕上打印“Hello!”。
- 执行 main 函数：在此例中，它不会做任何事（除了返回一个未知的整数外）。

main 后：删除所有的全局对象。在此例中，执行 TheWorld 的析构函数并打印“Goodbye!”。

这就是你的第一个面向对象的 C++程序!

但是，事情并非如此简单。如果编程真是如此简单，每个人都将成为一个程序员。要让程序运行必须向计算机输入一整套“神奇的咒语”，这样情况就变得复杂和困难了。正是这些“咒语”给我们的工作提供了安全性。



world1

现在是尝试编译和运行你的第一个 C++程序的时候了。你可以在光盘目录 src\world1 中找到对应的源代码和所需的项目文件。将此目录复制到硬盘上以供使用。温习“前言”结尾处介绍的如何使用编译器的技巧。对于书中的例子，只要光盘上提供了对应的源代码，我就在边注中说明源代码所在的 src 子目录。

还有一个提示：如果编译器拒绝编译你的程序（例如，如果你更改了源代码或使用了非标准兼容性的编译器），请查看它的输出。它会告诉你发现的错误及其位置。如果你理解这些错误信息，你可能已经是一位有经验的程序员了。如果不能理解，请不要绝望。只是有经验的程序员才能理解这些错误信息。

下面是针对非标准兼容性编译器用户的一些排错技巧：

- 编译器告知 main 中缺少一条 return 语句（发出警告）。编译器信息不对，但你也可以在大括号间添加 return 0;来消除此警告。
- 编译器不能找到头文件 iostream。编译器太旧，以致于不支持新标准库。尝试将 iostream 换成 iostream.h 并删除 cout 前面的前缀 std::。这是暂时的解决办法，你需要更新你的编译器。
- 程序打印 Hello!，但是不打印 Goodbye!。你的标准库版本有缺陷（不幸的是，Microsoft VC++的 6.0 版本有此缺陷）。幸运的是本书中的大部分例子并不需要此功能，所以不要有太多的担心。

在大部分情况下，成功构建程序后，你可以在调试器中单步运行它。你可以看到每条语句是如何执行的，以及它是如何改变程序状态的。参考编译器的说明文档了解如何使用调试器。

小结：至此我们已经介绍了类和对象。将类看作对象的蓝图。它描述了对象的结构和行为——在这里，就是构建和销毁对象时所发生的一切。一旦有了蓝图，你就可以基于它创建对象。

1.2　本地作用域

main 的作用域、给构造函数传递参数、整数类型。私有数据成员、初始化、嵌入本地作用域、for 循环。

world2

全局作用域是大括号外的所有区域。在执行 main 前，它被激活（即构建全局对象）；完成 main 后，它变成无效（删除全局对象）。另一方面，本地作用域为大括号内的区域（但在后面我们也会看到一些例外）。

本地作用域在程序流程进入时激活，在流程离开时失效。只要执行流程遇到本地作用域中对象的定义，它就得以构建；退出此作用域时，它就会被删除。这样的对象也叫自动对象（因为它们自动创建和删除），或叫做堆栈对象（因为它们占用了内存，即所谓程序的堆栈）。

本地作用域的第一个例子是创建一个本地 World，名叫 smallWorld，它位于 main 的作用域之内。

```
int main()
{
    World smallWorld;
    std::cout << "Hello from main!\n";
}
```

表 1-2 列出了源代码的对应打印输出。smallWorld 对象在程序进入 main 之后立即创建——此时，它的构造函数打印“Hello!”（这是第二个“Hello!”，第一个“Hello!”在构建全局对象 TheWorld 时打印）。

表 1-2　　打印输出及对应来源

打印输出	来　　源
Hello!	TheWorld 的构造函数
Hello!	smallWorld 的构造函数
Hello from main!	Main 中的输出
Goodbye!	smallWorld 的析构函数
Goodbye!	TheWorld 的析构函数

下一步，main 函数打印信息 Hello from main!。最后，在退出 main 的作用域之前，删除 smallWorld 对象。此时，它的构造函数打印 Goodbye!。

执行 main 并打印另一个 Goodbye!后删除全局 TheWorld。

world3

我将对原始程序进行另一个修改。我将使 World 的构造函数带一个整数（预定义的 int 类型）作为参数。此参数（也称为形式参数）将作为所创建的特定对象的标识符。在 World 的构造函数内，此参数 i 将被发送到标准输出（计算机屏幕）。

```
#include <iostream>

class World
{
public:
  World(int i)
  {
    std::cout << "Hello from" << i << ".\n";
  }
  ~World()
  {
    std::cout << "Goodbye!\n";
  }
};

World TheWorld(1);

int main()
{
  World smallWorld(2);
  std::cout << "Hello from main!\n";
}
```

请注意 std::cout 对象的聪明之处。它既接受由字符组成的字符串并作为字符串打印它们，也接受整数作为一个数字进行打印。而且，你可以看到，它可以使用多重<<符号逐个向 std::cout 发送一串参数。

因为 World 的构造函数现在需要一个参数，所以当我们创建此类的每个对象时，必须提供参数。在这里，我们只要在所创建的对象名后面的小括号中指定这一值即可。

```
World  TheWorld(1);
```

它看起来像调用一个函数，从某种意义上说，我们在调用构造函数——一个函数调用包括特定的函数名，其后面的括号中放一个值（或值列表）。这些值作为参数发送到函数。因为我们给 TheWorld 的构造函数传递 1，所以，它将打印 Hello from 1.（参见表 1-3）。

还有一些情况未说明。根据打印输出，我们可以说出哪个对象打印了第一个 Hello，哪个对象打印了第二个 Hello，但却不能辨别哪个对象打印了第一个 Goodbye!以及哪个对象打印了第二个 Goodbye!。如果传递给构造函数的数字确定了对象，这些对象仍然不能记住它们的身份。在析构函数中，我们不能只打印 i 的值，因为 i 不再存在。构造函数的形式参数 i 的作用域在构造函数的闭合大括号处结束。

表 1-3　　打印输出及对应来源

打印输出	来　　源
Hello from 1.	TheWorld 的构造函数
Hello from 2.	smallWorld 的构造函数
Hello from main!	main 中的输出
Goodbye!	smallWorld 的析构函数
Goodbye!	TheWorld 的析构函数

在我们解决此问题前，让我先讲解与之相关的另一个话题。我们在这里谈论的是面向对象的编程，但我还没有定义我所指的对象到底是什么。对象的定义如下：

定义：具有某种身份的事物就是对象。

如果你认为某事物不具有身份，那它就不是对象。“美丽”不是一个对象。但如果你说“此美丽与彼美丽不同”，你就赋予了这两个美丽以身份，于是，它们就变成了对象。我们有时定义具有非对象名字的类，这只意味着我们赋予这些旧名字一个新的“对象化”的意义。但是，一般而言，使用非对象名字并非上策。如果可能，你应该使用可数名词给对象命名（可以在它们前面添加不定冠词 a 或 an 的名词为可数名词，如“a world”、“a stack”、“an integer”等）。



world4

我不是说 World 对象没有身份，因为它们实际上具有身份，只是它们自己不能意识到。如果我们赋予它们记忆力，它们也能记住它们是谁。

```
#include <iostream>

class World
{
public:
  World(int id)
    : _identifier(id)
  {
    std::cout << "Hello from " << _identifier << ".\n";
  }
  ~World()
  {

 std::cout << "Goodbye from " << _identifier << ".\n";
  }
```

```
private:
  const int _identifier;
};
World TheWorld(1);

int main()
{
  World smallWorld(2);
  for(int i = 3; i < 6; ++i)
  {
    World aWorld(i);
  }
  World oneMoreWorld(6);
}
```

有几个新事物需要解释，让我以自顶向下的方式来讲解。在 main 中，我们创建了本地 World，名字叫 smallWorld，它带有一个参数，即对象的 ID，参数值为 2（但这一次，World 将在消亡前一直记住此参数）。

接下来，我们编写了一个 for 循环：

```
for(int i = 3; i < 6; ++i)
{
  World aWorld(i);
}
```

此特定 for 循环的意义如下：对于从 3 开始的整数 i，只要它小于 6，就在每次迭代后增加一个步长 1（表达式++i 表示给 i 的值增加 1），并完成如下任务：

1. 打开循环体的作用域。
2. 定义一个名叫 aWorld 的 World（它的 ID 等于 i 的当前值）。
3. 关闭循环体的作用域。

更准确地说，程序首先执行循环头的初始化部分：int i = 3。然后测试 i < 6，因为此条件为真（true），所以，它将执行循环体（如以上所解释的过程一样）。接着，它执行自增部分：++i，再次检查条件 i < 6 等。当自增操作使 i 变为 6 后，条件变成假（false），循环终止。

因为每次迭代都要打开和关闭循环体的作用域，所以 aWorld 的构造函数及析构函数每次都被调用。我们可以看到如下的打印结果序列：

```
Hello from 3.
Goodbye from 3.
Hello from 4.
Goodbye from 4.
Hello from 5.
Goodbye from 5.
```

循环完成其迭代过程后，World 调用定义为 oneMoreWorld 的名字。它的 ID 为 6，因而

我们将看到：

```
Hello from 6.
Goodbye from 6.
```

分隔 for 循环体的大括号可以省略，因为该循环只包含一条语句（这就是我前面所说的例外）。所以，我们可以按图 1-2 的方式书写此循环，而程序的执行方式完全一样。

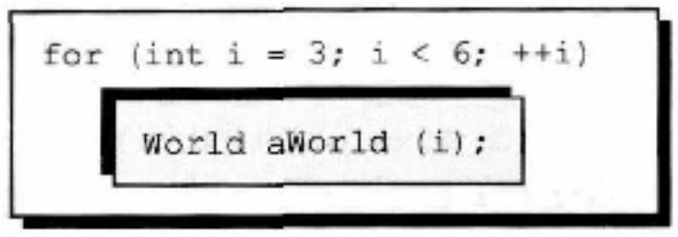


图 1-2　不含大括号的作用域（阴影部分）

for 的循环体形成了 main 作用域内的一个单独作用域。当程序进入此作用域时，来自外部作用域的对象不会被删除。事实上，如果没有名字冲突，它们仍然可见且可以访问。

如果将 smallWorld 改名为 aWorld，就会发生名字冲突，因为它将与 for 循环内的 aWorld 冲突。对于 C++来说，这是可行的；外部的 aWorld 只是在 for 循环内暂时不可访问。外部 aWorld 的名字将被内部 aWorld 的名字所隐藏。在下面内容中，我们将明白“可访问”的含义。

在 for 循环头中定义的变量 i 的作用域是什么？

```
for(int i = 3; i < 6; ++i)
{
  World aWorld(i);
}
```

这的确是一个不容易回答的问题。在老版本的 C++中，它的作用域常常扩展到循环的作用域之外。现在，情况并非如此（尽管许多编译器仍然支持这一不正确的解释）。在循环头中定义的变量的作用域即为循环本身。请注意，此作用域与循环体的作用域并不相同。在循环体中定义的对象在每次迭代时都重新创建。头中定义的变量在循环入口只初始化一次。然而，当离开循环时，两者都不可访问。

你不必借助于 for 循环引入一个本地子作用域。在一个现有本地子作用域的任何地方放置一对匹配的大括号后，实际就是创建了一个子作用域。作用域内定义的对象在控制流程离开时会被删除（当回答练习 1 时请考虑这一点）。

类 World 有一个数据成员：_identifier。

```
private:
  const int _identifier;
```

它的类型为 int，而且它是常量（const）和私有（private）的。为什么它是私有的？这与我在前面提过的数据隐藏有关。它意味着：今天此类的私有部分以这种方式实现，但明天——谁又知道呢？也许_identifier 将是一个 int，也许是一个字符串，也许根本不存在。类的用户不会关心这一点。如果_identifier 不是私有，它可能像 MS-DOS 中众人皆知的隐藏函数一样，客户端应用程序通过它们来启动，MS-DOS 的开发者为了兼容性不得不永远维护它们。

在 World 中并非如此！编译器将确保除了类 World 自身外没有其他任何人可以访问此私有成员。在 C++中，类定义分为公有和私有两部分。类的客户，也就是创建且与此类对象交

互的程序员，他们只能使用类定义中的公有部分内容。另一方面，类的实现者在实现类的过程中既可以使用公有部分，又可以使用其私有部分。在客户与实现者之间有着非常重要的区别。在此书中，你将看到许多这样的例子。

如果忽略关键字 public，则编译器当成缺省的私有[1]。如果你看到编译错误“Cannot access private members”（不能访问私有成员），你就有可能忘记了在类定义中指定公有部分。公有和私有部分可以按任意顺序分布，但我尽量遵循先公有部分、后私有部分的惯例。毕竟客户感兴趣的是公有部分，所以，我们将尽可能让它显而易见。

关键字 const 的意义是：没有人可以在对象的生存期间更改_identifier 的值。从物理上是不可能的（至少从“编辑上”上是不可能的）。但即使是一个 const 变量，也必须初始化一次。所以，编译器只提供了一次宝贵的机会，此时，我们才可以（事实上也必须）初始化 const 数据成员。这就是在构造函数的前导中，初始化常量数据成员甚至不能在构造函数内——只能在这一“前导”本身中进行。

前导是构造函数的一部分，它以一个冒号开头并且包含一个以逗号分隔的数据成员列表，每个数据成员后用小括号将初始化器括起来。前导_identifier 用 id（构造函数的形式参数）的值进行初始化。

```
World(int id)
 : _identifier(id)
{
   std::cout << "Hello from " << _identifier << ".\n";
}
```

一旦初始化_identifier 后，对于 World 特定实例的值就永远不会改变了（以后，我们将学习如何修改非常量数据成员）。当调用此对象的析构函数时，它将打印构造函数中传递给它的值。所以，当我们看到“Goodbye from 1.”时，我们知道这是 TheWorld 的结束（TheWorld 通过值为 1 的 ID 构建），与其他任何一个 aWorld 的结束相反。

为什么_identifier 的前面有一个下划线？这是一种约定。在本书中，所有私有数据成员的名字都以下划线开头，再写小写的字母（警告：以下划线开头的标识符及大写字母是库所使用的保留写法，所以，不要在代码中使用这种样式）。一习惯这种约定后，你就能借助它提高程序的可读性。当然，你也可以使用自己的约定。事实上，每个人都可以创建自己的约定（这种情况非常普遍）。

谈到约定——我绝对坚持大括号位置的约定，理由是：在本书所提倡的编程样式中，作用域扮演着重要的角色。让作用域在排版上突出显示对于提高程序的可读性至关重要。因此：

所有作用域的起始左括号必须与本作用域的闭合右大括号垂直对齐。

以前屏幕和打印机的省纸约定为：

```
for(int i = 3; i < 6; ++i){
    World aWorld(i);
```

[1] 在 C++中还有一个访问定义符：protected（受保护）。当我们在第 4 章讨论继承时，它才会发挥作用。

```
}
这种样式不利于程序的维护。
```

小结：对象可以有自己的记忆和状态。保存这种状态的数据成员更适宜声明为私有。最好在构造函数的前导中进行初始化，所以，对象一旦创建后就立即具有一致的状态。for 循环创建了自身的作用域。每次循环迭代都要进入和离开此作用域（这就是你看到以上那些“Goodbye from 3, 4, 5”信息的原因）。

1.3　嵌入对象

嵌入、嵌入初始化、构建/析构顺序。



world5

我们已经见过了 int 类型的数据成员——它是内置类型之一。C++的优点之一就是使内置类型和程序员自定义类型之间无实质性区别。当某些类的数据成员为用户自定义类型时，它就叫嵌入对象（embedded object）。在下面的例子中，Matter 被嵌入 World 中（我将赋予物质特性，例如，我可以说此物质比相邻宇宙中的彼物质更稳定）。另一种描述方式是：World 包含 Matter。

```
#include <iostream>

class Matter
{
public:
  Matter(int id)
    : _identifier(id)
  {
    std::cout << "  Matter for " << _identifier
              << " created\n";
  }
  ~Matter()
  {
    std::cout << " Matter in " << _identifier
              << " annihilated\n";
  }
private:
  const int _identifier;
};

class World
{
public:
```

```
World(int id)
  : _identifier(id),
    _matter(_identifier)//初始化嵌入体
{
std::cout << "Hello from world"
          << _identifier << ".\n";
}

~World()
  {
    std::cout << "Goodbye from world" << _identifier << ".\n";
  }
private:
  const int     _identifier;
  const Matter   _matter; // Matter类型的嵌入对象
};

World TheUniverse(1);

int main()
{
  World smallWorld(2);
}
```

此例中有意思的地方是 World 构造函数中的前导。

```
World(int id)
  : _identifier(id), _matter(_identifier)
```

它首先初始化_identifier，然后初始化_matter。初始化某个对象意味着调用它的构造函数。Matter 的构造函数需要一个整数，这是我们要传递的值。基于让_identifier 成为常量的同样原因，我们也让_matter 成为常量。它在 World 的生存期永远不会改变。它们在前导中初始化，而且永远不能改变，甚至不能被任何人访问。

顺便提及，//（双斜杠）用于注释语句。编译器忽略//后面的所有内容直到此行结束。

现在，我们感到奇怪的是初始化的顺序与前导中的初始化器顺序毫无关系。事实上，如果嵌入对象的构造函数不需要任何参数，它可以在前导中完全被忽略。如果不需要显式初始化，整个前导均可以被忽略。相反，初始化的规则是：

> 数据成员按它们在类定义中的出现顺序进行初始化。

因为在 World 的定义中，_identifier 出现在_matter 之前，所以+identifier 首先被初始化。这就是为什么我们可以在_matter 构造函数中使用它的值的原因。C++中嵌入体的顺序与语句的顺序同等重要。

嵌入对象按构建它们的相反顺序销毁。最后构建的对象将首先被销毁，依次类推。在面向对象将领域，对象嵌入称做“has-a”关系，例如，一个 World 有一个 Matter（记住，我们

已经对象化了物质）。此关系可以示意为图 1-3。

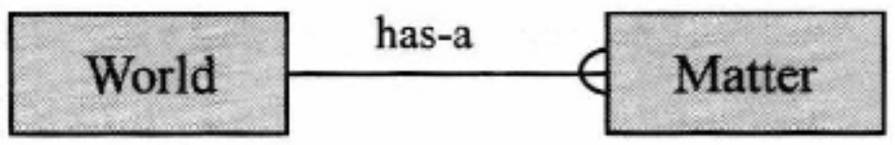


图 1-3　图示 has-a 关系

小结：程序员自定义对象类型可以作为数据成员嵌入其他类中。这些嵌入对象的构造函数在执行对象的构造函数体前被调用（从概念上讲，你可以将它们想象成在前导中被调用）。如果这些构造函数需要参数，它们必须在前导中初始化。构建顺序由嵌入对象的顺序决定。嵌入对象按构建它们的相反顺序销毁。

1.4　继承（Inheritance）

公有继承、基类的初始化、构建/析构顺序、double 类型。

对象间的另一种重要关系是“is-a”关系。当我们说恒星是一个天体时，我们的意思是恒星具有天体的所有属性，也许还有其他一些星体特有的属性。在 C++中，这种关系可以用继承来表达。我们可以说类 Star 继承了类 CelestialBody，或者说 CelestialBody 是类 Star 的基类（如图 1-4 所示）。

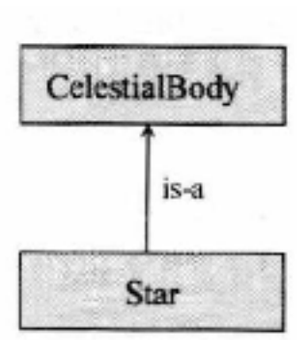


图 1-4　图示对象间的 is-a 关系

world6

请看下面代码：

```
#include <iostream>
class CelestialBody
{
public:
  CelestialBody(double mass)
    : _mass(mass)
  {
    std::cout << "Creating celestial body of mass "
              << _mass << "\n";
  }
  ~CelestialBody()
  {
    std::cout << "Destroying celestial body of mass"
              << _mass << "\n";
  }
private:
  const double _mass;
};

class Star: public CelestialBody // Star是一个CelestialBody
{
```

```
public:
  Star(double mass, double brightness)
     : CelestialBody(mass),
     _brightness(brightness)
  {
    std::cout << "Creating a star of brightness "
              << _brightness << "\n";
  }
  ~Star()
  {
    std::cout << "Destroying a star of brightness "
              << _brightness << "\n";
  }
private:
  const double _brightness;
};

int main()
{
  std::cout << "  Entering main.\n";
  Star aStar(1234.5, 0.1);
  std::cout << "  Exiting main.\n";
}
```

代码行：

```
class Star: public CelestialBody
```

告诉编译器 Star 继承了 CelestialBody 的所有属性。特别是因为 CelestialBody 包含_mass，所以 Star 也包含_mass。

CelestialBody 有一个需要 double 类型参数的构造函数。double 为一个内置类型，它对应与实数。它的全名是双精度浮点数（double-precision floating-point number），意思是它有一个小数点（浮点），而且它以两倍于单精度浮点（float）的有效位数来保存（注意，对象 std::cout 可以准确无误地打印双精度数）。

在 Star 的构造函数前导中，我们必须给 CelestialBody 的构造函数传递参数。我们完成这一点的方式是——使用它的类名调用基构造函数：

```
Star(double mass, double brightness)
     :CelestialBody(mass), _brightness(brightness)
```

构建的顺序非常重要。

首先完全构建基类，然后再构建派生类。

派生类的构建的通常顺序：首先是嵌入体、然后是构造函数中的显式代码。再次说明，前导中的顺序是无意义的，而且如果不存在显式初始化，前导中全部内容可以被忽略。析构的顺序也与构建顺序相反。特别是首先调用派生类的析构函数，然后再调用基类的析构函数。

1.5　成员函数和接口

输入流、成员函数（方法）、返回值、接口、返回void 的函数。

到目前为止，我们已经学习了对象的构建和销毁。我们还能对对象做些什么？我们不能访问私有数据成员，因为编译器不允许我们这样做。只有对象本身（如类定义中的一样）可以决定是否暴露它的某些行为和内容。通过成员函数（也叫方法）来暴露行为。如果成员函数为公有，它可以从程序的任何地方调用。

input1

我们的第一个成员函数 GetValue 返回一个 int。从函数的实现过程我们可以看到，返回值为私有数据成员_num（现在，我们知道可以访问某个对象的意义了：意味着可以调用它的成员函数）。

这里，InputNum 类型的对象 num 定义于 main 的作用域中，这正是我们可以访问它的地方。调用 num.GetValue()就是对它的访问。你也可以说，我们对对象 num 调用方法 GetValue()，或者说，我们正获取对象 num 的值。按 Smalltalk 的术语讲，我们给对象 num 发送消息 GetValue。我们得到的反馈是一个 int，我们用 std::cout 立即打印它。

```
#include <iostream>
class InputNum
{
public:
  InputNum()
  {
    std::cout << "Enter number ";
    std::cin >> _num;
  }

  int GetValue()const { return _num; }
private:
  int _num;
};
int main()
{
  InputNum num;
  std::cout << "The value is" << num.GetValue()<< "\n";
}
```

让我们弄清楚相关的语法。返回值的类型总是在成员函数定义的前面指定。一个需要有返回的方法必须在结尾有一个 return 语句（在后面，我们将看到在方法中途返回也是可行的）。如果函数没有返回，我们必须指定返回类型为 void。

在以前的所有例子中，声明 main 返回一个 int，但事实上它没有 return 语句。这是因为 main 的特殊性，编译器将提供缺失的 return。按习惯，如果程序成功运行，则 main 返回 0——而且这是由编译器生成的隐式返回值；如果发生错误，则可能从 main 返回一个“错误级别”——一个与程序有关的非 0 整数。

你可能会问：“返回给谁？”，返回给执行此程序的任何人。例如，如果你从一个批处理文件运行此程序，你可以使用一个批处理命令来检查错误级别，命令如下：

```
if not errorlevel 1 goto end
```

当从命令行（DOS 状态）执行此程序时，你可以在程序名后面添加参数列表。这些参数可以被程序的 main 读取。我们学习指针时，再返回来讨论这一问题。

现在讨论方法 GetValue 声明后面的关键字 const。

```
int GetValue()const
```

这意味着该方法没有改变对象的状态。编译器将不允许此方法的任何数据成员执行赋值、自增或自减操作，也不允许此方法调用这一类的任何非常量方法。而且，对于常量对象只能调用声明为 const 的方法。

到目前为止，我们见过了 World 中的 const 对象_matter。如果 Matter 有一个常量方法，如 GetValue，它可以由对象_matter 调用（即编译_matter.GetValue()）。以后，我们将会看到这样的例子。

注意，_num 不能再声明为 const，因为它没有在前导中初始化。构造函数体中的代码改变了_num 的（未定义）值。如果试图使用 const _num 来编译此程序就会得到错误信息。

当在 main 中构建类 InputNum 的对象 num 时，它提示用户输入一个数字，然后等待输入的数字保存在私有数据成员_num 中。此任务由输入对象 std::cin 完成：从键盘获取输入并保存在一个变量中。根据变量类型的不同，std::cin 要求从键盘获取不同的输入。例如，如果你用一个 int 来调用它，它就要求获取一个整数；如果你通过一个 double 来调用它，它就要求获取一个浮点数。

公有方法集称为对象的接口，它定义了客户与此对象交互的方式。例如，InputNum 提供了对其中内容的只读访问方式。因此，我们确保了 InputNum 的值一旦创建就永远不会改变。对 GetValue()的成功调用总是返回相同的数。

也可以对某个对象进行写访问——例如在本例中，我们定义如下方法来实现：

```
void SetValue(int i){ _num = i; }
```

然后，在下面的调用后：

```
num.SetValue(10);
```

后续对 GetValue 的调用将返回 10，不管在 num 构造函数中输入的是什么数。

C++中的等于号表示赋值。下面的语句：

```
_num = i;
```

将_num 的值设置成 i 的值。

当然，方法 SetValue 不能定义为 const，因为它改变了对象的状态（可以试一试）。而且

如果 Matter 有了这样一个非常量方法，此方法就不能在 World 的常量对象_matter 中调用。编译器严格控制这种情况的发生。

小结：公有成员函数集定义了一个给定类的对象的接口。更重要的是，一个设计优秀、实现完美的对象可以实现一种明确的“契约”。用 C++编程就是编写这样的契约并实现它们。一个接口要，也必须要按这样一种方式来设计：契约的客户部分简单且定义完善。对象甚至能抵御恶意客户对契约的破坏。我们讨论断言的使用时将继续探讨这一主题。

1.6　成员函数作用域

成员函数作用域、作为字符数组的字符串、调用其他成员函数。

input2

下一个例子演示了成员函数体如何形成自己的本地作用域。我们可以在此作用域内定义对象、创建子作用域等。

所以，这是输入对象中的一个新难点。

```
#include <iostream>

class  InputNum
{
public:
  InputNum()
  {
    std::cout << "Enter number ";
    std::cin >> _num;
  }

  int GetValue()const { return _num; }

  void AddInput()
  {
    InputNum aNum;  // 从用户处获取一个数字
    _num = _num + aNum.GetValue();
  }

private:
  int _num;
};

int main()
{
  InputNum num;
```

```
    std::cout   << "The value is "
                << num.GetValue()<< "\n";
    num.AddInput();
    std::cout   << "Now the value is "
                << num.GetValue()<< "\n";
}
```

在 InputNum 中添加了一个新的方法 AddInput。它使用了一个 InputNum 类型的本地对象来提示用户输入数字，此数字然后使用 GetValue 读取并添加到原始值上。

下面一系列图示演示了程序的执行过程。图 1-5 演示了对 main 中本地对象 num 的构造函数的调用。此构造函数提示用户输入数字并将它保存在数据成员_num 中。

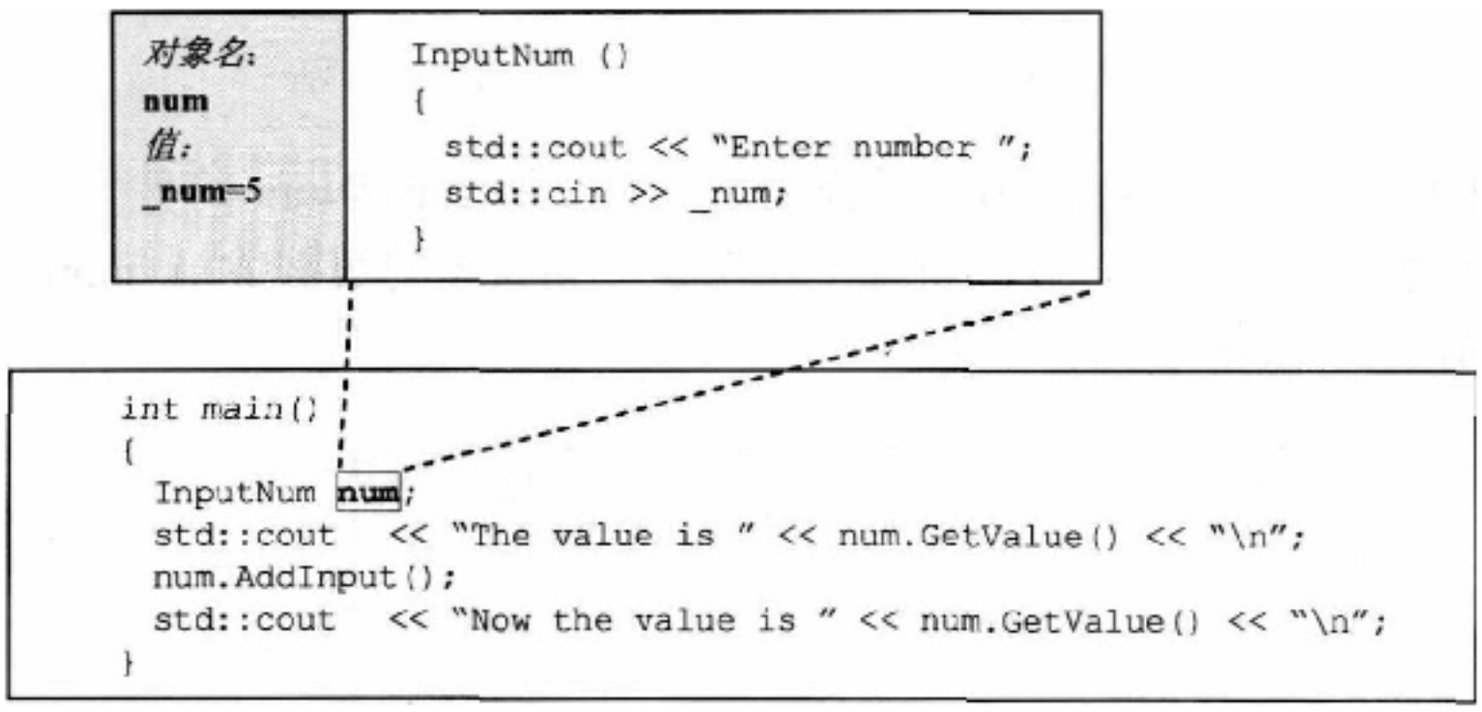


图 1-5 当执行 main 时，调用类 InputNum 的对象 num 的构造函数。它提示用户输入一个数。用户输入 5。这一值保存在 num 的私有数据成员_num 中

下面的图 1-6 演示了在 main 中对 num.GetValue()的调用。它返回在 num 的构造函数中所输入的值。

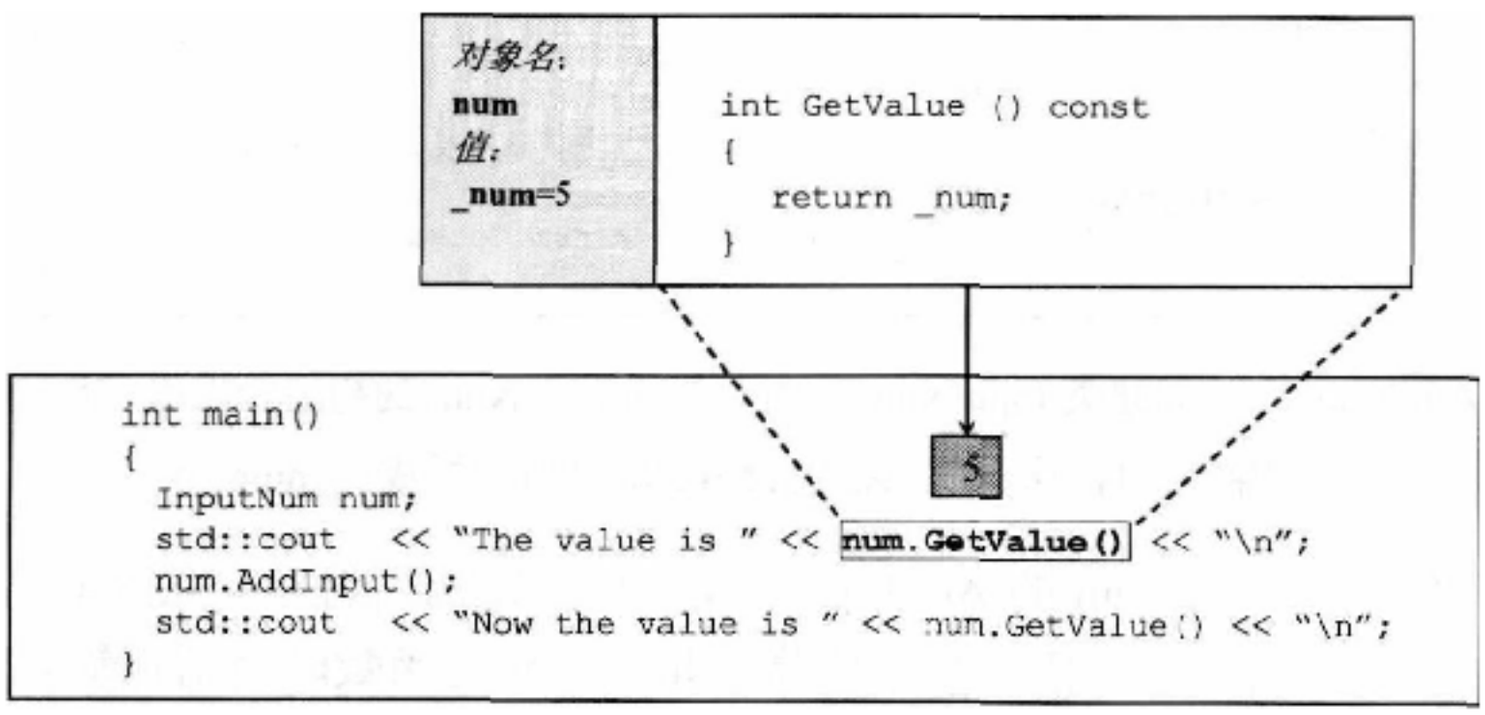


图 1-6 调用对象 num 的 GetValue 方法。它读取和返回保存在 num 的私有数据成员_num 中的值，这一值为 5

在接下来的图 1-7 中，我们可以看到相对于对象 num 调用 AddInput。保存在_num 中的第一个值与之相同。

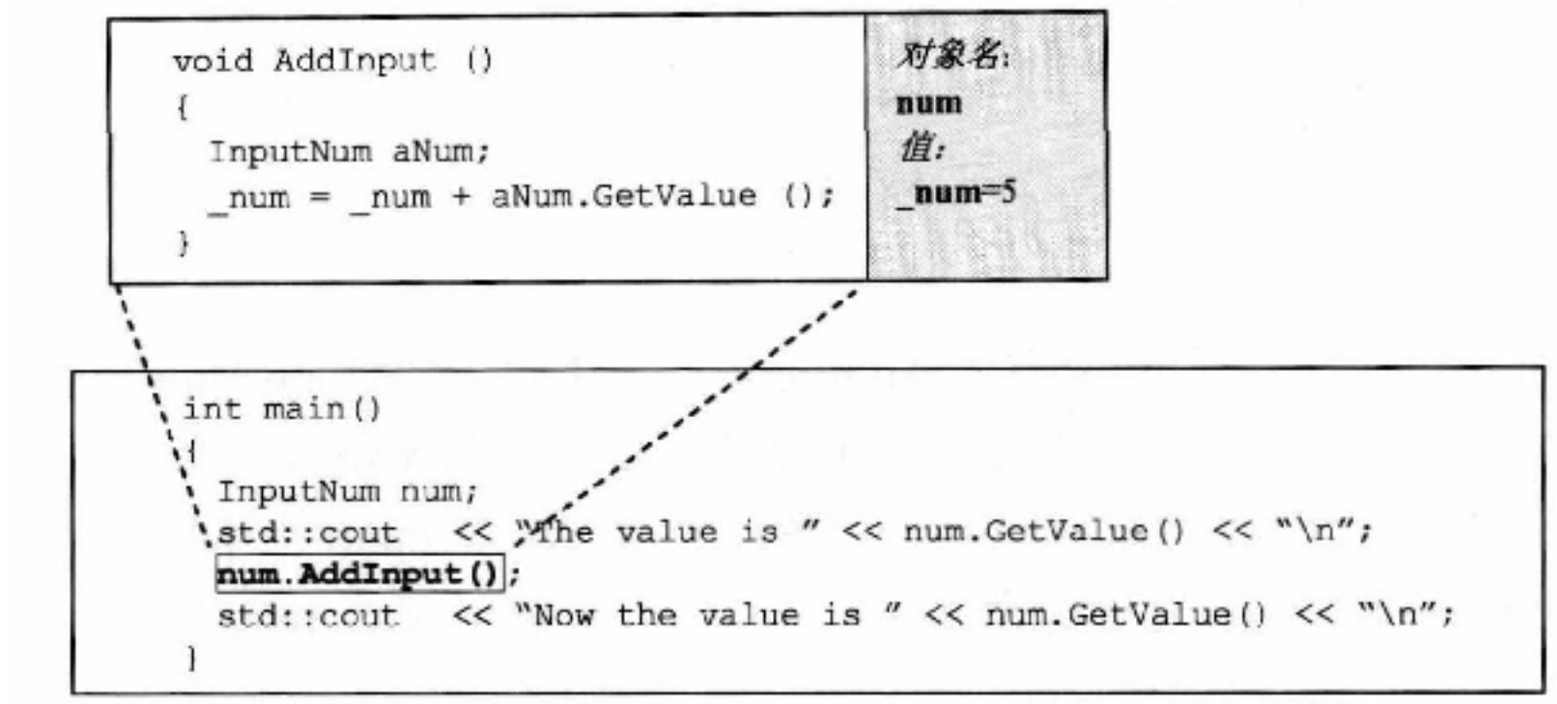


图 1-7　下一步调用对象 num 的 AddInput 方法

但是在图 1-8 中，在 AddInput 方法的作用域创建了类 InputNum 的另一个对象 aNum。它的构造函数提示用户输入另一个数，而且将它保存在 aNum 的数据成员_num 中。此_num 与对象 num 的数据成员_num 不同。

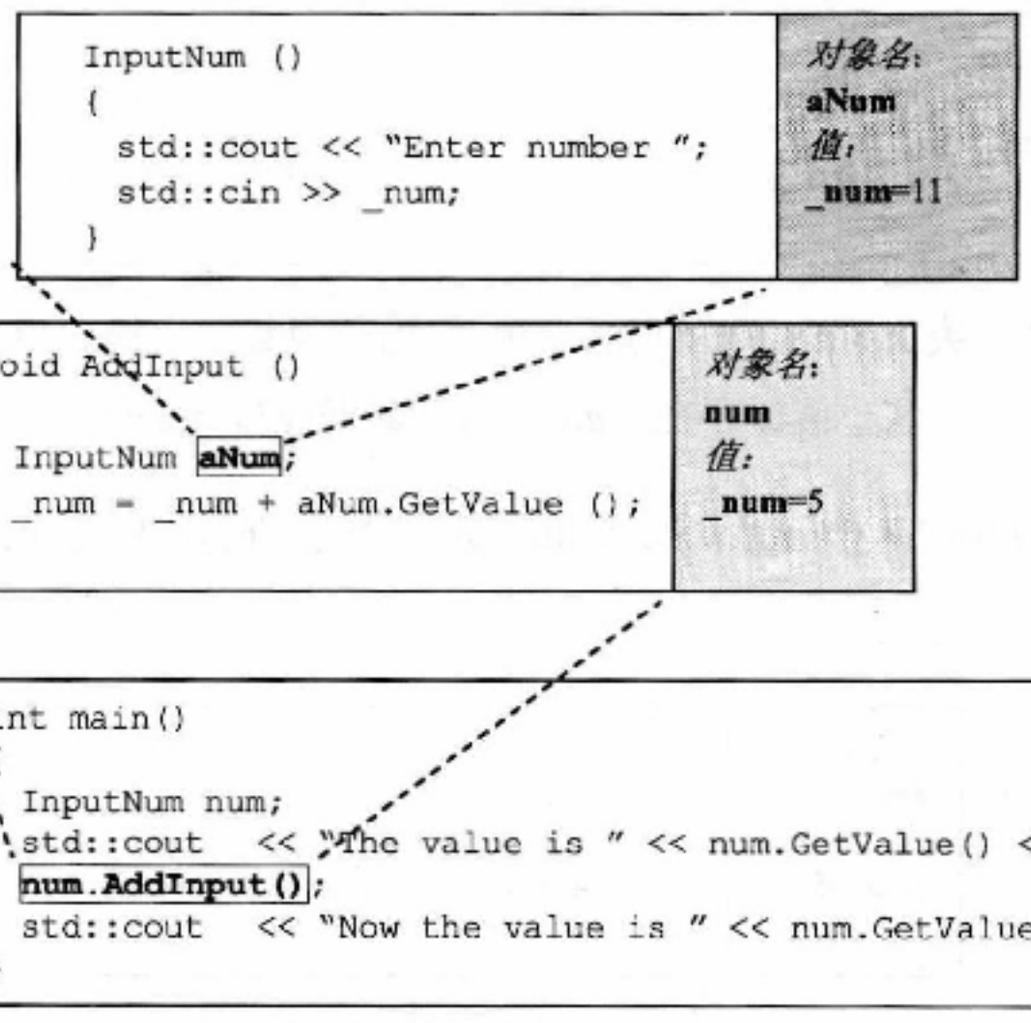


图 1-8　num 的 AddInput 方法创建类 InputNum 的对象 aNum。aNum 的构造函数提示用户输入一个数。用户输入 11，这一值保存在 aNum 的私有数据成员_num 中

根据图 1-9，仍然在对象 num 的 AddInput 方法中，程序调用对象 aNum 的 GetValue 方法。它返回保存在 aNum 中的值，此值正是在构建 aNum 的构造函数时由用户输入的。这一值被加到保存在对象 num 的数字上。

最后，图 1-10 演示了在 main 中，程序如何从 num 读取新增加过的值并打印出来。

你可能想知道程序如何执行一个方法，然后返回到调用者并从离开的地方继续执行。在

调用方法前，程序在哪儿保存它所执行任务的信息？使用程序（也称为执行）堆栈来保存。调用一个过程意味着在此（不可见）堆栈中推入一些数据。从此过程返回意味着从此堆栈中弹出这些数据并返回到前一个控制点和状态。

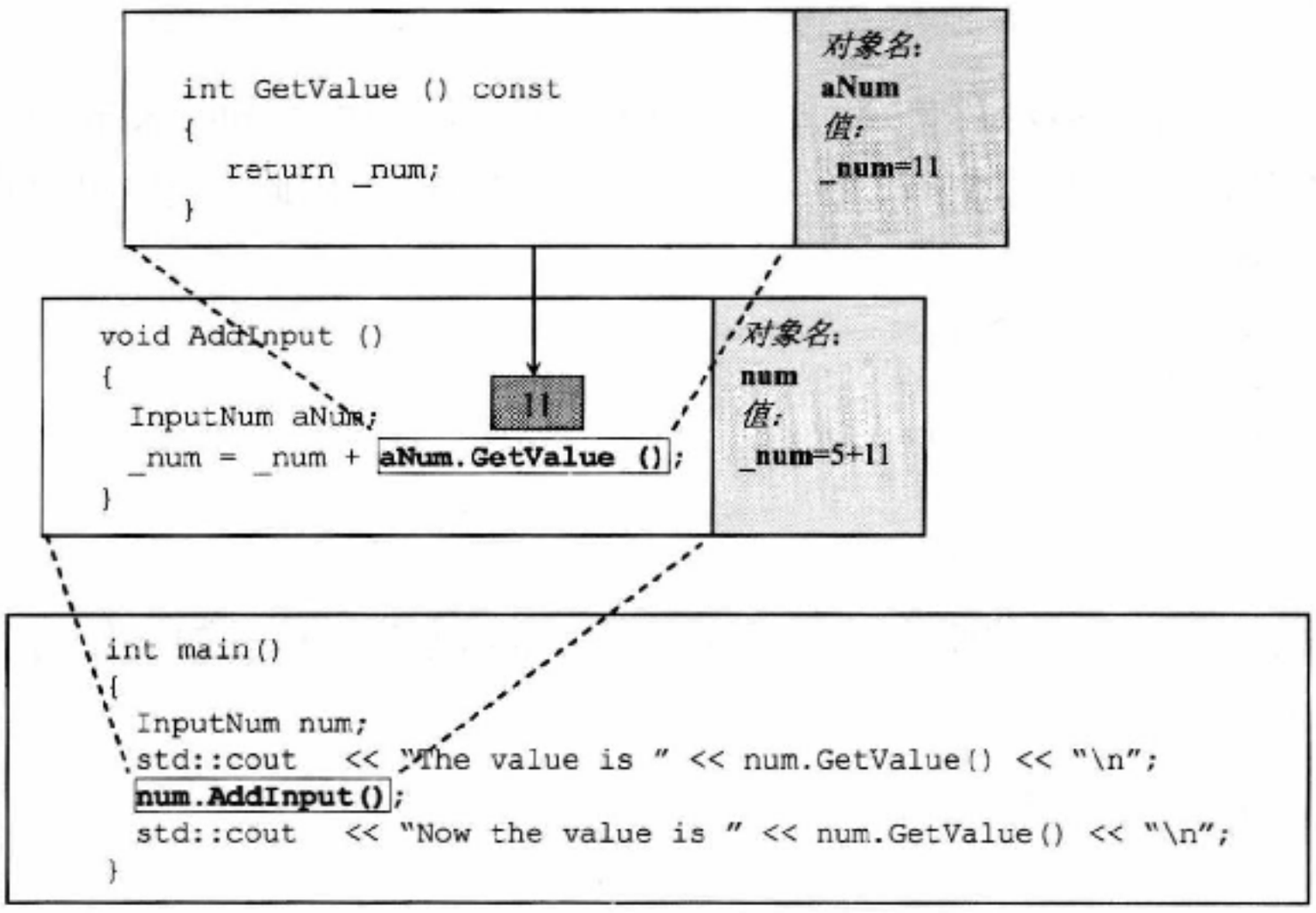


图 1-9　仍然执行 num 的 AddInput 方法时，调用对象 aNum 的 GetValue 方法。它读取和返回保存在 aNum 私有数据成员_num 中的值。然后，这一值 11 被加到 num 的私有数据成员_num 中。它的值为 5，加上 11 后，变成了 16

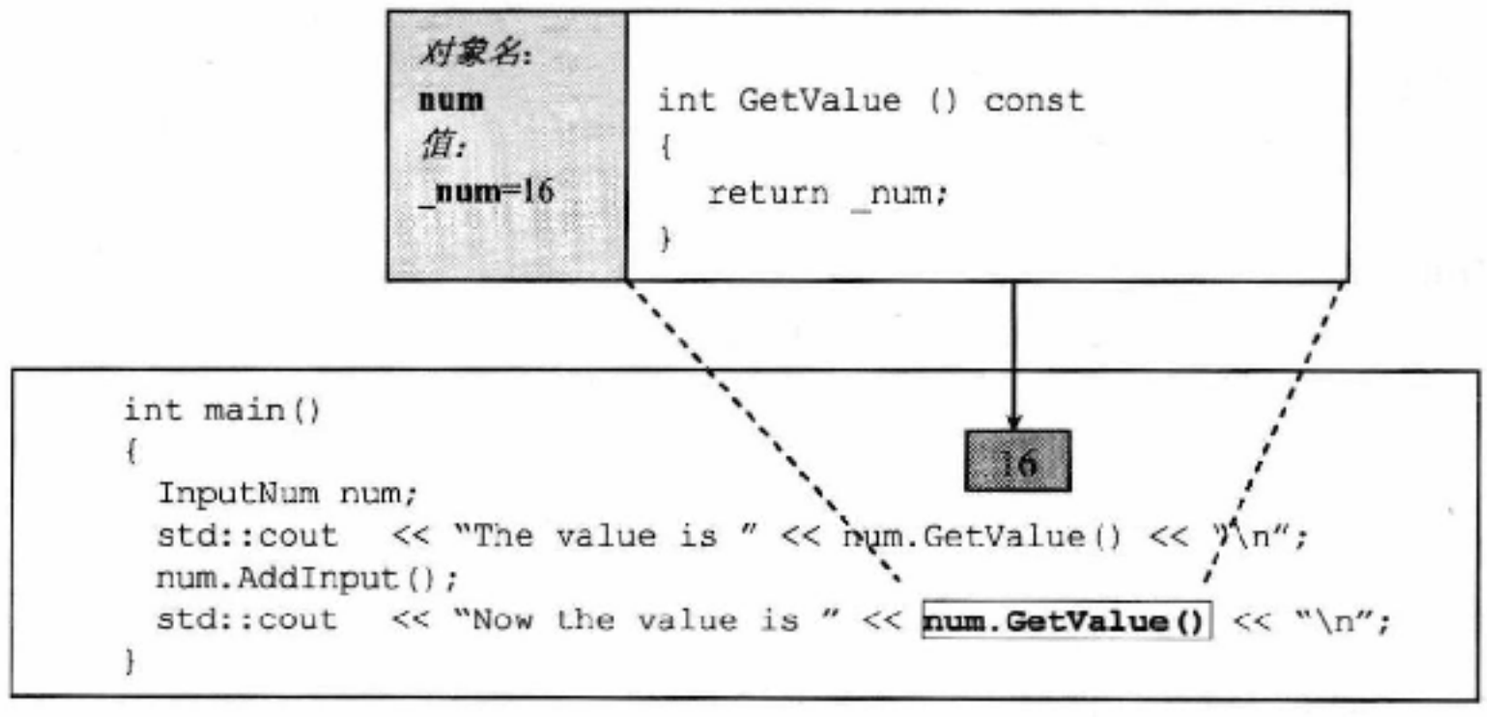


图 1-10　再次调用对象 num 的 GetValue 方法。它读取和返回 num 的私有数据成员_num 的值，现在它等于 16

特别是一个调用序列与这有些相似。首先，把即将传递给过程的所有参数推入堆栈中。接着把返回地址推入堆栈中——这是从调用返回后执行下一条指令的地址。然后，执行跳转到该过程的实际代码中。此过程开始时在堆栈中开辟一个保存本地变量的空间。如此一来，

每个过程调用都得到了所有本地变量的一个新拷贝。现在，执行此过程，然后弹出本地变量、返回地址及传递给它的参数，并跳转到返回地址。如果有返回值，则返回值通常通过处理器的寄存器之一进行传递。

请注意，这只是一个概念上的模型。实际执行细节与特定的调用约定有关，而且与处理器和编译器有关。

程序中有一处不太方便用户使用：总是出现相同的提示语“Enter number”。用户不知道此提示针对哪种情况。根据具体情况变化提示是不是更好呢？因此，我们可以在 main 中使用如下代码进一步完善：

```
InputNum num ("Enter number ");
num.AddInput ("Another one  ");
num.AddInput ("One more     ");
```

input3

InputNum 的一些方法必须带字符串参数。完全可以办到。请看如下代码：

```
#include <iostream>

class InputNum
{
public:
  InputNum(char msg [])
  {
    std::cout << msg;
    std::cin >> _num;
  }

  int GetValue()const { return _num; }
  void AddInput(char msg [])
  {
    InputNum aNum(msg);
    _num = GetValue()+ aNum.GetValue();
  }
private:
  int _num;
};

const char SumString [] = "The sum is ";

int main()
{
  InputNum num("Enter number ");
  num.AddInput ("Another one ");
  num.AddInput ("One more ");
```

```
    std::cout << SumString << num.GetValue()<< "\n";
}
```

在上面代码中有几个地方值得注意。它引入了一个新的内置类型：**char**。顾名思义，此类型用于保存字符。除非你使用的是一些实在很特殊的计算机，一个字符占用 8 位内存空间。

我们还知道字符串是一个字符数组——这就是字符串变量名后面方括号[]的意义。此例中有一点我们看不出来，即字面字符串以 0 结束；也就是说，在最后一个显式字符之后，编译器又添加了一个空字符（null）。例如，“Hi”实际上是一个 3 个字符的数组：‘H’、‘i’和 0（或者严格地讲是‘\0’）。空字符串与表示数字 0 的字符完全不同，后者写作‘0’。明白它们的区别了吗？空字符：0 或‘\0’；零：‘0’。因此我们知道，“\n”是一个由‘\n’和‘\0’组成的字符数组。终止空字符告诉操作于字符串的不同函数（包含前面例子中的 cout）此字符串在哪里终止。否则，函数将无从知晓。注意，字面字符串放在一对双引号中，而字符字面值放在一对单引号中。

顺便要说的是，如果不通过给输出发送字符串“\n”，你也可以通过发送特殊的 std::endl 得到相同的结果。语法如下：

```
std::cout << std::endl;
```

它的工作方式像变魔术（我的意思是，我现在还不想解释它）。你发送给标准输出或输入的这一特殊东西就称为操纵器（manipulator），它们的其中功能之一是修改格式。

在 C++的标准库中定义了一个特殊的 std::string 类型。如果你想操纵字符串完成一些任务，它将特别有用。我们将在以后学习它。

此程序另一个不同寻常之处是在 AddInput 内调用 GetValue 的方式。

```
_num = GetValue()+ aNum.GetValue();
```

我们进行了两个调用，一个调用没有指定对象，另一个调用对象是 aNum。我们知道，第二个是针对 aNum 对象调用 GetValue。第一个调用 GetValue 方法的对象是当前处理的对象（用 C++的术语讲就是“this”对象），而且只是读取_num 自己的值。不过，这是一种将我们分离于实现细节来完成的方式。这可能有点像无谓的重复（为什么我们不直接只使用_num 呢？），但在许多情况下，这将给将来的维护带来巨大的便利。

有些人可能想知道，调用一个成员函数而不直接读取_num 的值的开销有多大？或者会问，为什么不让_num 公有，然后在所有地方作为快捷方式使用它呢？访问一个公有数据成员的语法如下：

```
myNum._num
```

你也许不相信它，但这一调用实际上是任意的。我不是开玩笑——没有任何附加的操作。真的没有附加任何条件（以后讲解内联函数时，你就会明白其原因）。

因为使用成员函数访问数据成员没有运行时开销，所以没有太多理由在程序中使用公有数据成员。让数据成员公有化意味着释放控制权完全交给客户，但客户不仅可以使用该数据，还可以随意修改它。它还将你限定在特定的实现方式中。一个成员函数可以重实现，例如，根据

需要从一些外部源获取值。你可以给某函数添加额外的功能，例如，将每次调用记入日志或增加使用计数。如果不对客户代码进行大量的修改，使用公有数据成员就不能完成这一切。

小结：成员函数体形成了一个单独的本地作用域。构造函数和析构函数体也是如此。程序员可以在本地作用域内定义对象以及创建子作用域。

一个字符串是一个以空字符（null）终止的字符数组。数组用方括号与其名字来声明。字符数组可以用显式字面字符串来初始化。数组可以在全局或本地作用域内定义并作为一个参数传递给其他函数。

1.7　类型

内置类型、类型定义（typedef）。

下面简短描述内置类型：

- 布尔类型 bool 可以表示预定义值 true 和 false。
- int 是一个一般性的带符号整数类型。对于一个给定的机器，它是整数类型最有效的实现方式。它的大小与机器有关，但它至少有 2 个字节长，不过，通常至少是 4 个字节长。
- double 是一般性的带符号浮点类型，它对应于实数。
- char 是一般性的 8 位字符类型，它用于 ASCII 字符串。根据编译器的不同，缺省情况下可带符号或不带符号。
- wchar_t 是一个 16 位字符类型，它用于 Unicode 字符。
- short 和 long 是一种定义更具体的带符号整数类型。通常 short 为 2 个字节长、long 为 4 个字节长，但是在 64 位计算机上，它可能为 8 个字节长（与 int 同长）。如果你编写的程序只在一种类型的处理器上运行，你可以假定 short 和 long（以及 char）的大小。当你在可移动磁盘上保存数据或在网络上传输时，大小就变得至关重要。

 short 通常用作值域窄的 int（节省空间）。例如，在大型数组中，当 short 或 int 可能溢出时通常使用 long。记住，用 2 个字节只能保存从-32768 到 32767 之间的数（或者对于无符号类型来说，范围为 0~65 535）。算术结果可能很容易溢出此存储空间，此值将不知不觉被截断。
- float 是 double 的值域较窄版本（节省空间），但它的精度较低。在大型数组中使用它才有意义，因为这时大小有实质性的影响。
- long double 的精度比 double 更高。
- 所有的整数类型可以加上前缀关键字 unsigned 或 signed（所有类型缺省为 signed，除了如上所述的 char）。无符号的特殊情况如 unsigned char，它对应于某个机器字节；unsigned long，它通常用于低级的位逻辑操作。

当你厌倦了一遍遍地输入 unsigned long 时，C++允许你定义一个快捷方式，这就是 typedef

(类型定义)。这是一种给任何类型(内置或用户自定义)起不同名字的方式。例如,unsigned long 应用 typedef 命名为 ULONG:

```
typedef unsigned long ULONG;
```

定义此 typedef 后,就可以在使用其他任何类型的地方使用 ULONG,例如:

```
ULONG SwapValue(ULONG newValue)
{
  ULONG oldValue = _ulNumber;
  _ulNumber = newValue;
  return oldValue;
}
```

同样,你可以对 BYTE 使用 typedef:

```
typedef unsigned char BYTE;
```

与 C++的所有特征一样,typedef 可能被滥用。并不因为你可以用自己的名字重新定义所有的类型就意味你必须这样做。如果你在别人的代码里看到 INT 或 LONG,你会认为这是什么呢?是 int 或 long 的类型定义名字吗?如果是,为什么不直接使用 int 或 long?如果不是,那它到底代表什么呢?我无法想像出赋予如此基本数据类型 typedef 的合理原因,除非遵循的是某种不合常理的编码标准。

除了这些基本类型外,还有多种派生类型及用户自定义类型。根据以前讲过的字符数组,我们已经见识了派生类型,而且也见过了用户自定义类型,即类。

1.7.1 小结

任何类型(内置、派生或用户自定义)的对象均可以在全局作用域或本地作用域内定义。函数体(如 main)及成员函数形成了本地作用域。在一个给定的本地作用域内,程序员可以创建子作用域,子作用域还可以有子子作用域,如此等等。一个子作用域通常由大括号划定界限,但是在特定结构中(如 for 循环或条件 if/else),单条语句可能不带大括号,但仍然形成了一个子作用域(我们将在以后列举这样的例子)。

类是一个用户自定义类型。类行为由接口指定,接口是一套成员函数集(包括构造函数和析构函数)以及它们行为的描述。一个成员函数不带或带多个参数,并且返回一个值(也可以为空,即 void 返回类型)。类的实现由成员函数体及数据成员类型指定。实现过程完成由接口所声明的契约。

对象可以通过嵌入和继承组合在一起——从语义上讲与 has-a 和 is-a 的关系对应。嵌入和继承可以按不同的方式组合,导致对象中包含继承了其他对象的对象,此对象又继承了其他对象等。

1.7.2 练习

注意,如果你学习过 C,而且还没有被以上所讲知识吓倒,那么,下面这些练习对你来说似乎轻而易举。请务必完成这些练习。习惯于新的思考和解决问题的方式。所有的练习必

须只使用以上所介绍的结构来完成。

1. 使用类 World 编写一段程序，它的输出按顺序为“Hello from 1! Goodbye from 1. Hello from 2! Goodbye from 2.”至少找出两种不同的解决方案。

2. 找出下面类定义中的错误。给出两种纠正的办法。

```
class Glove
{
public:
   Glove(int numFingers)
     : _n(numFingers), _hand(_n)
   {
     std::cout << "Glove with " << _n << " fingers\n";
   }
private:
  Hand  _hand;
  int   _n;
};
```

3. 定义类 HorBar，它的构造函数打印如下模式：

```
+---------+
```

减号的个数作为一个 int 参数传递给构造函数。

同样，类 VerBar 打印一个高度为 n 的竖条，如下所示：

```
|
|
|
|
```

定义类 Frame，它包含（嵌入）上端的横条、竖条以及下端横条。它的构造函数带有两个参数：框架水平向的宽度和竖直向的高度，如下所示：

```
Frame(int hor, int ver)
```

Frame 对象可以使用下面的语法在程序中定义和初始化：

```
Frame myFrame(10, 2);
```

4. 定义类 Ladder，它包含两个框架及它们中间的一个 VerBar。构造函数要带两个参数，一个是水平向的宽度，另一个是水平条之间的垂直高度。通过（5,2）创建它时，必须打印如下模式：

```
+-----+
|     |
|     |
+-----+
|     |
|     |
+-----+
```

```
|
|
+-----+
```

5. 编写一段程序，它要求输入一个数字并计算其阶乘。n 的阶乘是从 1 一直到 n 所有正整数的积（包括 n）。n!= 1 * 2 * 3 *…* n。要将两个数相乘，就在它们中间放一个星号（*）。

6. 定义一个类 Planet，它继承于 CelestialBody，而且还另外具有反照率（表面的反射程度——使用一个 double 来保存它）。

7. 下面的类定义中，在构建对象时用实际的单词替换哑元字符串，打印字符串“program makes objects with class”。

```
class One: public Two
{
public:
   One()
   {
     Three three;
     std::cout << "dummy ";
   }
private:
   Four    _four;
};
```

Two、Three 和 Four 每个类在各自的构造函数中打印一些哑元字符串。

设计一个同样的类，在构建期间它将打印以上字符串，而且在析构期间按相反的顺序打印相同的单词。限制条件是，每个类在构造函数和析构函数中都必须具有完全相同的代码（打印相同的字符串）。

8. 设计、实现并测试类 Average。一个 Average 类型的对象应该接受输入给它的值。它应该能随时计算它们的累计算术平均值。

9. 设计一个类层次（继承链），从 LivingBeing 派生一个 Human。确保使用了 is-a 关系。现在，为 Tomato 和 Elephant 添加分支。编写一段简短的测试程序，在 main 内定义它们。每个类必须有一个构造函数来打印简短但有针对性的信息。

10. 设计一个 Human 来探讨 has-a 关系。一个人有一个头部，头部有一个鼻子，鼻子有鼻毛等。派生一个 Man 和一个 Woman。实现简单的类，它的构造函数打印信息，然后创建一对 Human。

1.8 抽象数据类型

抽象数据类型、包含文件、非内联成员函数、断言。

现在设计一些可以完成实际任务的有用类。让我们从一些简单、易于理解的知识点——

整数堆栈开始。一个堆栈由它的接口定义。你对堆栈可做的事主要就是推入一个数字以及弹出一个数字。堆栈之所以成为“堆栈”，是因为它的 LIFO（last in first out，后进先出）行为。你可以向堆栈中推入值，当你弹出它们时，它们是按相反的顺序退出。

通过描述它的行为，我们将堆栈定义为一种抽象数据类型。我们不必指定如何实现它，只是指定它要做什么。虽然我的描述不是十分精确或完全，但对堆栈也可以有一个正式的、独立于实现过程的数学定义。



stack1

我们选择一个整数数组来实现。这样可以用单独的文件很好地分离接口与实现的过程。在一些语言中，这实际上是可能的。在 C++中却并非如此（或者说，至少不通过一些技巧就不能实现）。大部分是技术上的原因。为了实现分离，编译器需要知道项目中的所有文件。[1]

在 C++中最佳的做法是将一些成员函数的实现代码分离到实现文件中。但是，接口文件必须包含此类的定义，而且包括指定所有的数据成员。根据传统，接口在头文件中定义，扩展名为.h。

下面是接口文件 stack.h 的内容。

```
const int maxStack = 16;

class Istack
{
public:
   IStack():_top(0){}
   void Push(int i);
   int  Pop();
private:
   int _arr [maxStack];
   int _top;
};
```

IStack 的构造函数初始化堆栈顶部索引为 0——即数组的开始。是的，在 C++中，数组元素的编号从 0 到 n-1，其中 n 为数组的大小。在这里，数组的大小为 16。它在语句中的定义如下：

```
const int maxStack = 16;
```

修饰符 const 告诉编译器，maxStack 的值永远不会改变。所以，编译器可以随时随地地用符号 maxStack 代替 16 这一实际值。

代码行：

```
int _arr [maxStack];
```

声明_arr 是一个 maxStack 整数值的数组。数组的大小必须是一个常量。正如我们在字符

[1] 例如，当编译器为对象分配内存时，必须知道它的大小。只有分析类的（公有以及私有）数据成员才能计算此大小。

组成的字符串中所见到的，当数组为显式初始化时，你就不必指定其大小。

注意，成员函数 Push 和 Pop 被声明但没有在类 IStack 的定义中定义。函数声明与函数定义的区别在于后者包括代码、实现过程，而且后面没有分号。前者只是指定了参数的类型和返回值的类型，即只指定了特征。那么，推入和弹出的实现代码在哪里呢？它们位于单独的实现文件 stack.cpp 中。

此文件的第一件事是包含头 stack.h 的 include 语句。下面我们包含新的头文件<cassert>。此文件包含了非常重要的函数 assert（实际上是一个宏）的定义。我不准备讲解它的详细实现过程——在编译时，通过定义符号 NDEBUG 就可以将此函数完全关闭。但是，只要我们没有定义 NDEBUG，此断言就会检查它的参数的逻辑真实性，即检查是否为一个非零值。换句话说，它断定它的参数的真实性。

我们在最后的程序完全测试后定义 NDEBUG，而且一次性地除掉所有的断言，以此提高程序的速度。许多编译人员区别对待调试版本和发行版本。当编译发行版本时，他们自动打开 NDEBUG 标记。

当断言的参数为假时（或等于 0 时）会发生什么呢？在这种情况下，断言将在程序的执行过程中被触发——它将打印一条指定源文件名、行号及不满足条件的信息。断言是一个调试工具。当一个断言失败时，它表示程序员出现一个错误——程序中存在漏洞。

我们并不希望出现漏洞——它们经常出现在意想不到的地方。不过，代码中存在一些可以捕获它们的焦点。我们在这些焦点作出假设。作出一定的假设是可行的——它们可使代码更简单、执行速度更快。但是，我们必须确保，至少在程序开发和测试期间，没有人破坏这些假设。

现在让我们看看 Push 和 Pop 的实现：

```
#include "stack.h"
#include <cassert>
#include <iostream>

//通过 NDEBUG=1 编译去掉断言
void IStack::Push(int i)
{
  assert(_top < maxStack);
  _arr [_top] = i;
  ++_top;
}

int IStack::Pop()
{
  assert(_top > 0);
  --_top;
  return _arr [_top];
}
```

值得注意的第一件事是，当成员函数的定义放到类定义的外部时，它的名字必须带有类名，还要在后面加上一对双冒号。

除此之外，还有一些相关问题要说明。目前为止，我们所定义的方法都是内联的。定义嵌入类定义中的成员函数由编译器自动变成内联。这意味着什么？这意味着编译器试图将函数的实际代码插入调用它的地方，而不是产生一个函数调用。例如，因为对象 input 的方法 GetValue 为内联，所以，它在

```
std::cout << input.GetValue();
```

中的调用从生成代码的角度看等价于

```
std::cout << input._num;
```

而另一方面，如果将成员函数的定义放到类定义的外面，就像 Pop 一样，它将自动成为非内联。

还可以作更多的变化：你可以将一个方法定义放到类定义之外，但仍然使之内联。要做到这一点，你必须在其定义的前面添加关键字 inline：

```
inline void IStack::Push(int i)
{
  assert(_top < maxStack);
  _arr [_top] = i;
  ++_top;
}
```

在这种情况下，你必须记住，当编译器编译调用此方法的代码时，它必须访问此定义（而不仅仅是访问类中的声明）。这就是为什么这样的定义通常放在由调用者包含的头文件中的原因。

到底应该使用内联，还是使用非内联方法呢？这取决于方法的复杂性。如果方法包含单条语句，让它内联通常使开销更少、代码更短。如果方法更复杂，而且它要从多个不同的地方进行调用，让它非内联更合理。

总之，内联函数对于 C++的程序编写至关重要。没有内联，实际上不可能让任何人信服 GetValue 和 SetValue 等方法的使用，而只是让_num 公有（甚至内联时，仍然难以让一些程序员信服）。

对数据的自由隐藏是 C++的一个巨大威力。在源程序中随便输入几行代码，甚至不必仔细斟酌，就可以轻松实现数据隐藏。

让我们更详细地分析堆栈的契约。首先，我们要求客户（正使用堆栈的程序员）跟踪堆栈中已有的元素，而且在任何时候都不要保存多个 maxStack 元素来防止堆栈的溢出。其次，当堆栈为空时，阻止客户调用 Pop，因此保证堆栈不会下溢。这些苛刻的要求由放置到适当地方的断言强制执行。

我并不是说，这是一个特别有用的整数堆栈实现方式——这些契约给客户增加了太多的负担。但至少这些契约是清楚的、可执行的，这是获得良好设计的第一步。

堆栈程序中的 main 函数为了演示它的工作方式，执行了少量的推入和弹出动作。

```
int main()
{
  IStack stack;
  stack.Push(1);
  stack.Push(2);
  std::cout << "Popped " << stack.Pop()<< std::endl;
  std::cout << "Popped " << stack.Pop()<< std::endl;
}
```


Stack 1

练习

11. 添加一个从堆栈的顶端返回值而不改变它的状态的方法 Top。再添加一个方法 Count，用于返回堆栈中元素的个数。

12. 修改 main 函数，让它违反堆栈的契约之一。看看当你打开断言时运行此程序所发生的一切。

13. 修改堆栈，让它变成 CharStack——字符堆栈。使用它来颠倒字符串的顺序：推入字符串所有的字符，然后逐个弹出并打印它们。

14. 设计一个双精度的抽象数据类型 Queue，它包含方法 Put 和 Get，并遵循 FIFO（先进先出）原则。用一个数组和两个索引（put 索引以及跟踪 put 索引的 get 索引）来实现它。特定队列的读出契约为：在 Queue 的生存期间，Put 方法的调用次数永远少于 maxPuts 次。当 Queue 为空时不能调用 Get。

15. 运用下面方法设计和实现 DblArray 数据类型：

```
void Set(int i, double val)
```

及

```
double Get(int i)const
```

设置和读取数组中索引为 i 的特定单元值。添加另一个方法

```
bool IsSet(int i)const
```

如果此单元从未被设置过，则此方法返回 false，否则，返回 true。契约为：当索引大于 maxCells-1 时不调用 Set 和 Get；不读取尚未设置值的单元，已经设置值（写过一次或读过多次）的单元不能再设置值。数组可以保存 double 类型的任何值。

提示：如果你想对一个布尔结果取反，可以在它的前面加一个惊叹号（!）。例如：

```
!IsSet(i)
```

当此单元从未设置，则为 true；反之为 false。

第2章 数组和引用

在程序中通过对象的名字来标识一个对象。但是如果我们在使用它们时，在所有地方都必须通过它们的名字来调用每一个对象的话，程序中的对象名将会泛滥成灾。程序将处于一个庞大的无结构名字空间之中。在不同的作用域给同一个对象起不同的名字提供了一种编程的间接级别。在计算机科学里有一句谚语：每个问题均可以通过间接方式解决。这种间接可以使用引用来实现——一个别名或一个可选名，每次不同的对象进入一个指定的作用域时都可以添加此名字。

计算机擅长于完成那些需要耐性和重复性的任务——它们的耐性比人类大得多。让人抄写“I will not challenge my teacher's authority”一百遍简直是一种折磨。但告诉计算机执行此任务一百遍，它会不折不扣地完成。这就是迭代的威力。

2.1 引用

引用对象、友元、按引用传递、初始化引用。

到目前为止，我们已经学习了对象间的 has-a 和 is-a 类型关系。第三种关系类型就是 has-access-to 关系。has-access-to 关系的不同之处是它不是“析构性的”。A 可以访问的对象在 A 消亡时不会消亡。

在 C++中，以引用的形式可以得到 has-access-to 关系的最单纯表达式。一个引用指向一个对象。它就像一个现有对象的临时名字——一个别名。它可以像原始对象一样使用；访问引用的语法与访问对象的语法一样。程序员可以传递引用，返回引用，甚至在其他对象中保存它们。无论何时，只要对象的内容更改，对此对象的所有引用都会更改。反过来，任何引用都可用于修改它所引用的对象（除非它是一个常量引用，这将在后面讲述）。

声明引用变量的方法是在类型名后面加上一个&符号。实际上，&符号与变量名捆绑使用，所以，如果你想在同一条语句中声明多个引用，你必须在所有的名字前面均加上&：

```
Istack & stack;
```

一旦创建了引用并初始化为引用某特定对象，那么它将总是引用此对象。给引用赋值并不会使它“指向”另外的对象；它用新值覆盖它所引用的对象。例如，对于下面的代码：

```
int i = 5;
int j = 10;
int & iref = i; // iref 指向 I
iref = j;
```

执行最后一条语句时将 i 的值从 5 改成了 10。它不会使 iref 指向 j。无论怎样都不能使 iref 指向 i 以外的其他对象。

不初始化引用就不能创建此引用。如果一个引用保存在某对象中，它必须在此对象构造函数的前导中初始化。

stack2

我们将在一个堆栈序列生成器的定义和实现中使用引用。序列生成器是作为项目序列用于表示其他一些对象的对象，它逐个返回保存在此对象中的值。

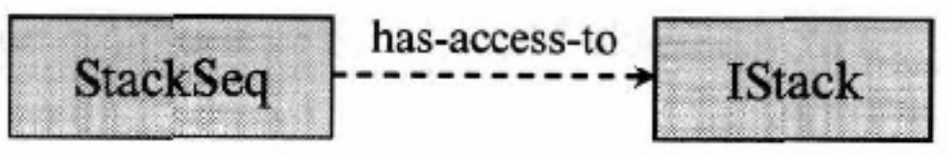


图 2-1 图示对象间的 has-access-to 关系

为了从此堆栈中回读值，序列生成器需要访问堆栈的私有数据成员（如图 2-1 所示）。堆栈可以使序列生成器成为自己的友元而授权对它的访问。堆栈的所有私有数据成员仍然对于其余对象不可见，仅有它的新友元堆栈序列生成器例外。请注意，友元关系是这样一种关系：StackSeq 可以访问 IStack 的私有数据，但不允许有其他行为。

IStack 和 StackSeq 的结合是一种较大粒度结构的例子，这是将几种数据结构统一到一个函数单元的一种模式。

```
class Istack
{
  // 让 StackSeq 可以访问 IStack 的私有数据成员
  friend class StackSeq;
public:
  IStack(): _top(0){}
  void Push(int i);
  int Pop();
private:
  int _arr [maxStack];
  int _top;
};

class StackSeq
{
public:
  StackSeq(IStack const & stack);
```

```
  bool AtEnd()const;   // 是否完成?
  void Advance();      // 移到下一项
  int GetNum()const;   // 读取当前项
private:
  IStack const & _stack; // 引用 stack
  int           _iCur;  // stack 当前索引
};
```

StackSeq 的契约声明，如果 AtEnd 为 true，客户不能调用 Advance 或 GetNum。

通过一个对常量 IStack 的引用调用 StackSeq 构造函数。数据成员之一也引用一个常量 IStack。常量引用不能用于更改它所引用的对象。例如，因为 Push 不是一个常量方法，编译器不允许 StackSeq 调用_stack.Push()。它也不允许在此数组中进行写入操作，或者更改 _stack._top。

对常量的引用可以用其他任何引用来初始化，但是反之则不行——非常量引用不能通过一个常量的引用来初始化。因此，我们可以给 StackSeq 构造函数传递任何堆栈。与构造函数的声明一致，编译器将让堆栈改成对一个常量 IStack 的引用。然后，_stack（它同样也是一个对常量 IStack 的引用）必须在 StackSeq 构造函数的前导中初始化（记住，引用必须在前导中初始化）。

数据成员_iCur 保存了序列生成器的状态——在所引用堆栈内对数组_arr 的索引。sequencer 是类 IStack 的友元，它完全知道堆栈的实现过程。

```
StackSeq::StackSeq(IStack const & stack)
  : _iCur(0), _stack(stack)//初始引用
{}
```

剩下方法的实现逻辑比较直接易懂。

```
bool StackSeq::AtEnd()const
{
  return _iCur == _stack._top; // 友元: 可以访问_top
}

void StackSeq::Advance()
{
  assert(!AtEnd()); // 非结尾
  ++_iCur;
}

int StackSeq::GetNum()const
{
  assert(!AtEnd());
  return _stack._arr [_iCur]; // 友元: 可以访问_arr
}.
```

注意，通过引用访问对象数据成员的句点语法与直接访问对象的语法相同。变量_stack 是相对堆栈对象的一个（只读）别名。因为 sequencer 为友元，所以 StackSeq 可以自由访问

IStack 私有数据成员。

此程序还引入了新逻辑运算符。惊叹号从逻辑上对它后面的表达式取反。当 sequencer 没有执行时，!AtEnd()为 true；反之，!AtEnd()为 false。你可以读作“非结尾”。

双等于号==用于相等性测试。如果两个数量不等，则结果为 false；如果相等则为 true。

警告：赋值运算符=和相等性运算符==间的相似性是导致严重编程错误的根源。

非常不幸的是，因为 C++对 C 的继承性，这样的一个印刷错误就可以导致几乎无法察觉的编程错误。不要指望编译器能找出如下语句

```
x == 1;
```

或下面的条件

```
if(x = 0)
```

中的错误（一些编译器将发出警告，但只是高级别的警告）。

根据相同的逻辑，不等于关系使用运算符!=来表达，如 x != 0。如果 x 不等于 0，则此表达式为 true；如果相等，则表达式为 false。

main 过程测试序列生成器的功能性。它推入几处数字到堆栈中，然后重复遍历它们。

```
int main()
{
  IStack TheStack;
  TheStack.Push(1);
  TheStack.Push(2);
  TheStack.Push(3);

  for(StackSeq seq(TheStack);
      !seq.AtEnd();
      seq.Advance())
  {
    std::cout << "  " << seq.GetNum()<< std::endl;
  }
}
```

请注意，对象 TheStack 是如何传递给序列生成器的构造函数的。编译器根据类 IStack 定义中的构造函数的声明知道程序员的所指为引用。因为构造函数声明为带有一个引用，所以将传递此引用（不是整个对象的拷贝）。

当一个方法的形式参数定义为一个引用时，我们说它是按引用方式传递的。当你按这种方式调用一个方法时，你传递的是对一个变量的引用，而且此方法可能会更改此变量的值。这与按值传递参数相反。当你调用一个希望按值方式获得一个变量（没有&）的方法时，此方法不会更改此变量的原始值。方法只操作它的拷贝。

比较下面两个方法。

```
class Dual
{
```

```
  public:
    void ByValue(int j)
    {
       ++j;
       std::cout << j << std::endl;
    }

    void ByRef(int & j)
    {
       ++j;
       std::cout << j << std::endl;
    }
};

int main()
{
  Dual dual;
  int i = 1;
  dual.ByValue(i);
  std::cout << "After calling ByValue, i = "
       << i << std::endl;
  dual.ByRef(i);
  std::cout << "After calling ByRef, i = "
       << i << std::endl;
}
```

第一个方法没有更改 i 的值，但第二个改变了 i 的值，从此程序的输出可以看出：

```
2
After calling ByValue, i = 1
2
After calling ByRef, i = 2
```

你是否担心从调用者的角度不容易看出使用的是不是一个引用——没必要!一般来讲，假定按引用传递对象、按值传递内置类型（而不像以上例子一样）。只是在一些特殊情况下才按值传递对象，原因也相当明显（参见后面章节“按值传递”中的内容）。所以，当我们阅读大部分程序时，对于传递给函数的对象，你可以放心认为它是按引用传递的。

让我们跟踪 main 中引用的执行路径。

```
void main()
{
  IStack TheStack;
  TheStack.Push(1);
  TheStack.Push(2);
  TheStack.Push(3);

  for(StackSeq seq(TheStack);
       !seq.AtEnd();
```

```
        seq.Advance())
    {
      std::cout << "  " << seq.GetNum()<< std::endl;
    }
}
```

以 stack 的别名——StackSeq 构造函数的形式参数，把对 TheStack 的引用传递给 StackSeq 的构造函数。

```
StackSeq::StackSeq(IStack const & stack)
  : _iCur(0), _stack(stack)// init reference
{}
```

这是对象 TheStack 的一个新临时名字。此引用不能用于修改 TheStack，因为它声明为 const。接下来，在 StackSeq 构造函数内部，我们初始化对象 seq 的引用数据成员_stack。再次说明，此引用不能用于修改 TheStack。从对象 seq 被创建一直到它被销毁，_stack 都将是 TheStack 的一个只读别名。序列生成器借助于只读别名_stack 来识别 TheStack。

注意 for 循环的初次使用。我们让 for 循环头包含一条初始化语句、一个测试表达式及一条后循环动作语句。第一条语句——循环初始化——只在进入循环前执行一次。在这里，它包含序列生成器的定义。下面的表达式——循环条件——在每次进入循环体时进行测试。如果表达式为真（非 0），迭代就开始（或继续）。当表达式为假时，循环退出（或从不进入）。在此例中，通过测试条件来判断序列生成器是否位于结尾处。第三条语句——循环自增——在每次迭代后执行。在这一例中，它推进了序列生成器。请注意，在正常退出循环后，循环条件总是以 false 值结束。

for 循环头三个部分的任何一个部分都可以为空。例如，一个无限循环通常为：

```
for(;;)// 永远执行
{
  // 这里略去了其他语句!
}
```

你可以使用 break 或 return 语句退出此循环。

小结：引用总是其他一些变量的别名（或者严格地讲是所谓 l-值的别名）。修改引用（例如给它赋值）更改了它所引用（或“指向”）变量的内容。“读取”一个引用，例如，在表达式中被动地使用引用就会得到它所引用变量的值。常量引用只能用于读取。

引用在创建之时必须初始化，或在其声明中完成；或对于成员引用，在构造函数的前导中完成；或者调用一个带有引用的方法时，在调用之时完成。

2.2 运算符

我们已经见过许多作用于数字和变量上的不同操作。我们知道，在两个数中间放一个加号（按程序员的叫法就是加法运算符）就可以将这两个数相加。加法只是算术运算的一个例子。

你可以使用相等性或不等性运算符来比较数字，这就是逻辑运算（或布尔运算）。运算

的结果为 true 或 false。

最后，还有一些作用于单个位的运算——它们称做位逻辑运算。

2.2.1　算术

在 C++中，你可以对数字和变量进行标准算术运算。表 2-1 列出了基本的算术运算符。

表 2-1　　基本算术运算符

符　　号	二元运算符	示　　例
+	加	a+b
-	减	a-b
*	乘	a*b
/	除	a/b
%	模除	a%b
	一元运算符	
+	正	+a
-	负	-a
	赋值运算符	
=	赋值	a=b

其中的一些运算符需要作一些解释。例如，为什么表中有两个除运算符？

首先，如果你对一个浮点数进行运算（例如对 double 类型数字进行运算），你必须使用常规的除运算符/。即使只有两个数或变量之一为小数，如 3.14 或一个 double，相除的结果将是一个 double 类型的数。但是，如果两个操作数都是整数，即没有小数点的字面数或定义为 int 的变量（或者其他整数类型），此时应用运算符/所得到的结果就是商的整数部分。例如，3/2 得到 1，它是 1.5 的整数部分。同样，10/3 得到 3，如此等等。

%运算符只可以应用于整数类型，它得到商的余数。例如，3%2（读作“3 除以 2 的余数”）等于 1。同理，11%3 得到 2，如此等等。

下面表达式总为真（只要 b 不为 0）：

```
(a / b) * b + a % b = a
```

但是当其中一个数为负数时，使用运算符/和%的结果无定义（虽然仍满足以上等式）。所以，当你运算负数时，不要使用它们。认为这是不利的一方面是有道理的。认为应该在 C++中实现此行为的论据并不充分：与 C 兼容，而且能通过处理器指令有效实现。

一元加没有实质性的作用。它只是一元减的对称运算符，一元减的作用是让数字的符号相反。所以，如果 a=2，那么，–a 得到–2，+a 得到 2。

2.2.2 逻辑

表 2-2 包含比较和相等性测试运算符列表，它们的运算结果为布尔值。它们大部分用于 if 语句中，并作为循环中的终止/继续条件。你也可以使用它们初始化布尔变量。

表 2-2 生成布尔值 true 或 false 的运算符

符　　号	相等性测试运算符	示　　例
==	等于	a==b
!=	不等于	a!=b
	比较运算符	
>	大于	a>b
<	小于	a=	大于或等于	a>=b
<=	小于或等于	a<=b

你可以使用逻辑与（AND，符号为&&）、或（OR，符号为双竖线||）组合简单的逻辑表达式。你还可以在一个逻辑表达式前面加一个惊叹号对它取反。表 2-3 列出了这些运算符。

表 2-3 逻辑运算符

符　　号	逻辑运算符	示　　例
&&	与	a && b
\|\|	或	a \|\| b
!	非	!a

记住有关运算符&&和||的一件重要事情：如果运算符左边的值决定了整个表达式的值，就不用再对右边求值。例如，假如 x = 0，在下面的表达式中：

```
if(x > 5 && x < 10)
  std::cout << x << " is between 5 and 10, exclusive\n";
```

因为第一个不等式 x > 5 已经为 false，所以，不会测试第二个不等式 x < 10。一个假的表达式用 AND 与其他任何表达式相连得到的结果总是 false。这种情况下，没有必要对第二个表达式求值，C++程序将忽略此过程。因此上面的代码等价于下面的代码：

```
if(x > 5)
```

```
{
  if (x < 10)
    std::cout << x << " is between 5 and 10, exclusive\n";
}
```

这样不仅会导致运行速度加快（第二个表达式可能具有相当的复杂性），而且还确保了在第一个表达式为 false 时，第二个表达式不会产生任何副作用（例如，如果你用++x < 10 代替第二个表达式，你将会看到当 x 小于 5 时，它不会自增）。

同样，如果逻辑 OR 的左边为真，其右边不会求值，因为一个真的表达式用 OR 与其他任何表达式相连时，得到的结果总是 true。

2.2.3　位逻辑运算符

在 C++中，你可以对整数执行几种类型的位运算（参见表 2-4）。二元位运算符 AND（与）和 OR（或）分别用符号&和|表示。其用法如下所示：

```
int i = 3; // 二进制 0011
int j = 5; // 二进制 0101
int k = i & j; //二进制 0001
k = i | j; //二进制 0111
k = k | 8; //二进制 1111
```

位补码用一个波浪号~表示。例如，在一个 4 位计算机上，如果 k=13（二进制 1101），那么，它的补码 j=~k 将等于 2（二进制 0010，1 变成 0、0 变成 1）。

异或运算符与不进位的位加法相似。如果你 XOR 两个 1 位，结果就是 0 位——没有将 1 进位到下一个位置。例如，按二进制方式表示的异或：0011^0101 = 0110。

表 2-4　　位运算符

符　　号	二元位运算符	示　　例
&	位与	a&b
\|	位或（OR）	a\|b
^	位异或（XOR）	a^b
>>	右移位	a>>n
<<	左移位	a<<n
	一元位运算符	
~	位补码	~a

右移位运算符对有符号与无符号整数的工作方式不同。当作用于无符号类型时，它向右移位，舍弃右溢出的位，并用 0 填充左边的空位。例如，以二进制表示的 1011 >> 1 = 0101。

但对于有符号类型并非如此——在此情况下，填充位等于原始数最左边的位。所以，如果例中的数字是一个有符号类型(而且我们使用的是一台 4 位计算机)，则 1011 >> 1 结果为 1101。如果你知道计算机是如何保存负数的，你就会明白其意义：它们总是将最左边的位设为 1。所以，向右移位不会改变它们的符号。事实上，对正数和负数来说，右移一位总是等价于除以 2。同样，左移一位等价于乘以 2（假定此数不会溢出）。

2.3 基于堆栈的计算器

自顶向下设计、制作（make）、私有成员函数、if 语句、do/while 循环。

我们下一步将利用目前所学的技术设计和实现一个小型但有用的程序。它是一个简单的基于堆栈的计算器。虽然这是一个非常微小的项目，但我们遵循设计和实现过程的所有步骤。我们将从外部规范开始，然后处理框架设计，最后实现此程序。

2.3.1 外部规范

基于堆栈计算器有一个简单的界面。它接受用户数字形式和运算符形式的输入。数字保存在 LIFO 类型内存中——将它们推入堆栈。对从堆栈中弹出的两个顶级数字应用二元运算符，运算的结果再次推入堆栈的顶部。如果堆栈中只有一个数，它就用两个操作数代替。每次动作完成后均显示整个堆栈的内容。

因为计算器实现了所谓的“反向波兰表示法”（reverse Polish notation，或称做后缀表示法，postfix notation），所以不需要小括号。为简单起见，我们只实现 4 个基本的算术运算。运行结果示例如下（用户输入为>提示符后面内容，我的注释放在小括号中）：

```
> 3(推入 3)
3
> 2(推入 2)
3
2
> +(相加 2 + 3)
5
> +(相加 5 + 5)
10
```

2.3.2 设计

顶级对象显然是计算器本身。它保存了内存中的数字并对它们进行运算。因为我们在每次动作完成后都要显示计算器堆栈的内容，所以，我们需要访问堆栈序列生成器。最后，为了完全分离计算器的输入/输出操作，我们必须用一个 input 对象从用户处获取输入，并进行一些预处理（符号化），便于区别数字与运算符。

对于输出，我们将使用标准的 cout 对象。

下面的列表描述了这些对象的最短生存期：

- 计算器在整个过程中必须有效，因为它有一个长期的内存。
- 输入对象与单个输入操作关联。
- 序列生成器的作用域与每个堆栈显示操作（其后是一个成功的计算器动作）关联。

输入对象从标准输入获得一个字符串。它可以区分数字与运算符，根据输入返回不同的符号。如果符号是一个数字，就将字符串转换成一个 int，并且让输入对象记住其值。

计算器对象接受输入对象，从中可以得到预处理数据，并执行相应的命令。如果数据为一个数字，计算器将它保存到堆栈；如果数据是一个运算符，则计算器执行此运算。创建一个堆栈序列生成器来显示其结果，并打印它的内容。为了迭代，计算器只是授予堆栈访问权限。

注意，我们在顶层只使用了三种交互作用的对象类型。还有一种类型，即堆栈，我们可以将它看作一个黑匣子（我们不调用它的任何方法，只是从一个组件到另一个组件传递对它的访问权限）。这不仅仅是此项目简单性的一个反映——我们必须总是尽量应用少量的顶层对象。一旦创建了此顶层，我们就可以开始自顶向下程序:设计，自顶向下地设计下层对象。我们现在进入更深一层的设计，直接开始设计计算器对象。我们已经知道计算器有一个堆栈，而且此堆栈对象将嵌入其中。我们将重用先前设计的堆栈。

2.3.3　简短的实现

自顶向下实现过程遵循自顶向下的设计思想。基于我们的框架规范，我们可以开始编写 main 过程。

```
int main()
{
  Calculator TheCalculator;
  bool status;
  do
  {
    // 输入提示
    std::cout << "> ";
    Input input;
    status = TheCalculator.Execute(input);
    if(status)
    {
      for(StackSeq seq(TheCalculator.GetStack());
                      !seq.AtEnd();
                      seq.Advance())
      {
        std::cout << "  " << seq.GetNum()<< std::endl;
      }
    }
  } while(status);
}
```

这里介绍了一些新的结构：do/while 循环及 if 语句。只要 while 子句的条件一直为真，do/while 循环体就重复执行。请注意，与 for 循环体不同，do/while 循环体总是至少执行一次。

还有一个版本的 while 循环，它在执行循环体前检查条件。事实上，任何循环都可以改写成一个 while 循环。例如：

```
for(int i = 0; i < n; ++i)
{
  std::cout << i << "\n";
}
```

等价于：

```
{
  int i = 0; // 初始化
  while(i < n)// 条件
  {
    std::cout << i << "\n";
    ++i; // 自增
  }
}
```

包围整个结构的大括号是限制循环变量 i 所必需的。

像在 for 循环中一样，while 循环体形成了一个单独的本地作用域（即使它只有一条语句而且省略了大括号）。

只有 if 子句中的条件（小括号内的内容）为真时，才进入 if 语句体，否则一起忽略。条件可以直接得到一个布尔值 true，就像在本例中一样，或者得到一个非 0 的其他值。一个非 0 值自动转换成布尔值 true。

在 if 子句内使用对数字求值的表达式而不用布尔值被认为是一个不良方式——即受到 C 的遗传，在 C 中就没有布尔类型。记住，你可以总是将一个数字与 0 比较将此数字转换成一个布尔值。例如，不要使用下面简洁的语句：

```
if(x)
```

你应该使用下面可读性更强的语句：

```
if(x != 0)
```

下面是一个在 if 语句中使用算术表达式结果的反面例子：

```
if(x = y)
```

这类代码编译器可以接受，因为在 C 和 C++中，赋值是一个表达式，它产生一个值——这里，此产生值与 y 的值相同（赋值后也与 x 的值相同）。如果你必须使用赋值作为 if 子句的一部分，至少要让代码更加明显一些，如：

```
if( (x = y) != 0)
```

再次说明，即使 if 语句体只有一条语句——在此情况下大括号可以省略，它也形成了一个本地作用域。

还请注意，变量 status 定义时没有进行初始化。在 C++中，我们要尽量避免此类情形。这里，我没有初始化变量，因为它总是在 do/while 循环体内进行初始化（而且我们知道，它至少要执行一次）。我不能在循环作用域内定义此变量，因为它在属于外部作用域，在 while 子句中被测试。每次迭代时，循环体执行之后都要对 While 子句求值。

下面，我们将根据自顶向下方式，为所有类编写简短的实现代码。这些简短的代码不会实现所期望的功能，但它们将让我们勾画了程序的顶级结构。而且，我们能编译和运行此“草图”并查看顶层流程控制。

堆栈从计算器传递到序列生成器，而且我们不需要知道它的任何细节（与 stack 为类 IStack 的一个对象不同）。所以，临时试验实现代码如下：

```
class IStack { };
```

序列生成器中省去了所有的方法。我决定用一个哑元变量_done 来模拟堆栈的定义。GetNum 返回随意指定的数 13。

```
class StackSeq
{
public:
  StackSeq(IStack const & stack)
    : _stack(stack), _done(false)
  {
    std::cout << "Stack sequencer created\n";
  }
  bool AtEnd()const { return _done;}
  void Advance(){ _done = true; }
  int GetNum()const { return 13; }
private:
  IStack const & _stack;
  bool    _done;
};
```

在此级别细节下，类 Input 仅暴露它的构造函数：

```
class Input
{
public:
  Input()
  {
    std::cout << "Input created\n";
  }
};
```

计算器再次包含一个哑元变量，它的目的是在 main 中执行一次迭代后跳出循环。

```
class Calculator
{
public:
  Calculator(): _done(false)
  {
```

```
  std::cout << "Calculator created\n";
}
bool Execute(Input& input)
{
  std::cout << "Calculator::Execute\n";
  return !_done;
}
IStack const & GetStack()//常量
{
    _done = true;
    return _stack;
  }
private:
  IStack _stack;
  bool _done;
};
```

方法 GetStack[1]返回一个对 IStack 的常量引用。换句话说，GetStack 为计算器的私有对象_stack 准备了一个只读别名，而且让调用者可以使用它。用户可以使用此别名访问_stack，但只是通过它的常量方法；或者，如果它是 IStack 的一个友元，可以通过读取_top 和保存在数组_arr 中的值来访问它。这正是序列生成器所需要的。

还应该注意，编译器将 return_stack 语句解释成返回一个对_stack 的引用。这是因为 GetStack 声明为返回一个引用。如果 GetStack 声明为：

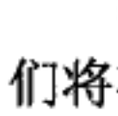
```
IStack const GetStack();
```

编译器将返回堆栈的一个只读拷贝。拷贝 stack 的开销比提供一个对它的引用稍高。我们将在后面介绍值类时再讨论此问题。

放置所有的哑元后，我们可以编译并执行此测试程序。它的输出正与我们的预料一致。

```
Calculator created
> Input created
Calculator::Execute
Stack sequencer created
  13
> Input created
Calculator::Execute
```

2.3.4 实现



calc2

因为我们的项目将日益增大，我们必须将它分解成单独的文件。类定义需要 3 个头文件——

[1] 必须指出，我对 GetStack 方法使用 const 修饰符，因为在这一临时实现中，GetStack 修改成员_done（更理想的技术是声明_done 时添加 mutable 修饰符，这将在后面讨论）。

calc.h、stack.h 和 input.h，以及 3 个实现文件——calc.cpp、stack.cpp 和 input.cpp。要从多个源文件中生成一个可执行程序，我们需要连接器的帮助。幸运的是，在集成环境中，你可以将所有文件添加到此项目中，然后让集成环境生成所需程序。不过，这里不想对它作太多的探讨。

头文件越多，我们就越可能两次包含同一个文件。这种情况是如何发生的呢？假如我们在 calc.h 内包含了 input.h，然后我们又在 calc.cpp 内包含了 calc.h 和 input.h，由此就可能带来麻烦。编译器将看到类 Input 的两个定义（它们相同，但无影响）。防止此情况发生的途径是为头文件添加条件编译指令。让我们将整个 input.h 用下面的指令对包围：

```
#if !defined input_h
...
#endif
```

如果没有定义符号 input_h，编译器将处理这两个预处理指令（它的作用正如它的名字所暗示的一样）中的代码。在开始，没有任何人定义此符号。所以，第一次编译器遇到#include "input.h"指令时，它将继续去处理 input.h。不过，if/endif 对中的第一个指令定义了符号 input_h：

```
#define input_h
```

下一次编译器碰到#include "input.h"指令时，它就会略过#if ! defined input_h 和#endif 之间的所有代码，因为 input_h 已经被定义（惊叹号是我们熟悉的逻辑非）。

Calculator：实现代码

开始一个自顶向下的过程，第一个实现的就是 Calculator 自身。当实现计算器时，我们将从 Input 对象找出需要得到的东西。Execute 方法应该从输入获得以下符号之一：数字符号、算术运算符符号之一或错误符号。对每一种符号类型执行动作均不同。对于数字符号，我们从输入读取数字值并推入堆栈；对于运算符，我们弹出两个数字（如果没有两个数，则弹出一个数），将它们传递给 Calculate 方法，并推入计算结果。

```
bool Calculator::Execute(Input const & input)
{
  int token = input.Token();
  bool status = false; //假定失败

  if(token == tokError)
  {
    std::cout << "Unknown token\n";
  }
  else if(token == tokNumber)
  {
    if(_stack.IsFull())
    {
      std::cout << "Stack is full\n";
    }
    else
    {
```

```
    _stack.Push(input.Number());
      status = true; //成功
  }
}
else
{
  // 契约: Input不能产生任何其他符号
  assert(token == '+' || token == '-'
         || token == '*' || token == '/');
  if(_stack.IsEmpty())
  {
      std::cout << "Stack is empty\n";
  }
  else
  {
      int num2 = _stack.Pop();
      int num1;
      // 特例: 仅当堆栈中有一个数
      // 对两个操作数使用这个数
      if(_stack.IsEmpty())
          num1 = num2;
      else
          num1 = _stack.Pop();
          _stack.Push(Calculate(num1, num2, token));
          status = true;
      }
    }
    return status;
}
```

我们已经介绍了条件的扩展——else 子句。只有当前面的 if 语句没有执行时（因为条件为假），才执行 else 语句体。

if/else 语句可以按如下方式排列：

```
if(A)
  ...
else if(B)
  ...
else if(C)
  ...
else
  ...
```

Calculate 是 Calculator 的一个新方法。因为它只能在 Calculator 的实现代码内使用，所以将它声明为私有。它带有两个数字和一个符号，而且返回运算的结果。我将此代码分离到一个方法中，因为 Execute 方法变得太庞大了。Calculate 有一个精心定义的功能，完全不受其

余代码的影响[1]。它用一个较大的交错排列的 if/else 结构来实现。

```
int Calculator::Calculate(int num1,
                          int num2,
                          int token)const
{
  int result;

  if(token == '+')
    result = num1 + num2;
else if(token == '-')
  result = num1 - num2;
else if(token == '*')
  result = num1 * num2;
else if(token == '/')
{
  if(num2 == 0)
  {
    std::cout << "Division by zero\n";
    result = 0;
  }
  else
    result = num1 / num2; // 整数除
  }
  return result;
}
```

请注意其中字符字面值（如'+'、'-'、'*'、'/'）的使用。与字符串字面值不同，字符字面值放在单引号中。不用给运算符符号赋予特殊值，我们只需使用它们的字符字面值（ASCII 值）。

让我们看看修改后的 Calculator 类定义。

```
class Calculator
{
public:
  bool Execute(Input const & input);
  // 让 stack 可以访问
  IStack const & GetStack()const { return _stack; }
private:
  int Calculate(int n1, int n2, int token)const;
  IStack _stack;

};
```

我们现在可以添加原来的 IStack 和 StackSeq 实现代码，添加方法 Token 和 Number 的哑元实现代码来扩展哑元 Input，定义 tokError 和 tokNumber 为常量整数，重新编译并运行

[1] 严格来说，Calculate 应该是一个静态（static）方法，因为它没有使用任何 Calculator 成员。我们将本书后面章节讨论静态方法。

此程序。

Input：实现代码

现在实现 Input 类。



calc3

```
#if !defined input_h  // 防止多重包含
#define input_h

const int maxBuf = 100;

// 符号为 tokNumber, tokError, +, -, *, /.

const int tokNumber = 1;
const int tokError = 2;

// 从 stdin 获得输入，转换成符号

class Input
{
public:
  Input();
  int Token()const { return _token; }
  int Number()const;
private:
  int _token;
  char _buf [maxBuf];
};

#endif // input_h
```

注意方法 Token 和 Number 声明为 const。Calculator 是只读访问 Input 类（通过一个常量引用），而且仍然可以调用这些方法。缓冲器_buf 是保存从用户获得的字符串的地方。

文件 input.cpp 的实现代码中包含两个新标准头。

```
#include <cctype>
#include <cstdlib>
```

头 cctype 包含（非常有效）宏的定义，它可以识别字符类型。例如，如果字符为从 0 到 9 的数字之一，宏 isdigit 就返回 true；否则返回 false。另一个文件 cstdlib 是声明函数 atoi 所需的文件，此函数的作用是将一个表示小数的 ASCII 字符串转换成一个整数。C++ 标准库中包含有更多这样的函数，你可以参阅编译器的在线帮助或印刷手册（或其他书籍）。

```
Input::Input()
{
  std::cin >> _buf;
```

```
// 输入的第一个字符通常容易判断是什么符号

int c = _buf [0];

if(std::isdigit(c))
  _token = tokNumber;
else if(c == '+' || c == '*' || c == '/')
  _token = c;
  else if(c == '-')// 允许输入负数
  {
    if(std::isdigit(_buf [1]))// 参照下一个字符
      _token = tokNumber;
    else
      _token = c;
  }
  else
    _token = tokError;
}
```

Input 的构造函数从标准输入读入一行文本到字符缓冲器。这是由友元 cin 完成的另一个巧妙任务（顺便说一下，到目前为止，我们尚未考虑缓存溢出的情况，我们仍然还处于“业余编程”水平）。

根据缓冲器中的第一个字符，我们决定使用何种符号。注意应用了减号的特殊语句，它可能是一个二元减，也可能是一个数字前的一元减。为了确定到底是哪种运算符，我们要参照缓冲器中的第二个字符。请注意我前面所提过的字符分类宏的使用。当字符为数字时（即0、1、2……9）isdigit 返回 true。字符分类宏使用一个非常有效的表查找方法来实现。调用 isdigit 实际上比执行如下执行代码更有效（&&是逻辑与——AND，参见前面“逻辑”中的有关内容）：

```
if(c >= '0' && c <= '9')
```

头 cctype 还包含除了 isdigit 以外的其他有用宏，如 isspace、islower 和 isupper。

如果符号为 tokNumber，则 Calculator 需要知道它的值。方法 Number 使用 cstdlib 中声明的库函数 atoi 将缓冲器中的字符串转换成 int 值。

```
int Input::Number()const
{
  assert(_token == tokNumber);
  return std::atoi(_buf);  //将字符串转换成整数
}
```

2.3.5 练习

1. 在计算器中添加位运算。
2. 添加一个新的命令 x，它用于将十进制和十六进制输出间的相互转换。

提示：为了以十六进制（以 16 为基数）输出一个数，对输出应用操纵符 hex。

```
std::cout << std::hex;
```

为了将输出换回十进制，则应用操纵符 dec。

3. 创建一个“逻辑计算器”，它用于处理布尔值。处理对象不是数字，它对应于 true 和 false 接受 t 和 f。它执行逻辑与、或、非以及接受对应的符号&、|及!（注意，“非”是一元运算，它带一个参数）。在打印布尔值前，对输出流加载操纵符 std::boolalpha。

第3章 指针

使用引用，我们可以赋予同一个对象多个名字。使用指针，我们可以让同一个名字指向不同的对象——指针是一种多变引用。

通过指针，我们可以创建复杂的数据结构。它们还增强了我们自由发挥的能力。一个指针就像一个插头，它可以插入插座。如果你有太多的插头和太多的插座，它们所连接的电线将会混乱不堪。程序员不得不在创建试验电路板似的程序与印制电路似的程序间寻求平衡点。

3.1 指针的不利之处

指针变量保存了一个对象的内存地址（例如，此对象为一个变量、一个数组、一个函数、某个类的一个实例等）。使用指针可以访问它所指向对象的内容。从这一点上讲，指针的行为像一个引用：它是其他一些对象的别名。但是，在指针和引用之间存在一个巨大的区别。指针并不永久依附于对象——它们可以删除。程序员可以让指针换成指向另一个对象（或根本不指向任何对象。）这是一个非常强大的功能，同时，也是一个非常危险的功能。大部分编程的 bug 在不同程度上与指针的使用有关。C++（及 C）有以下几种指针：

- 未初始化指针；
- 指向解除分配存储单元的指针；
- 越出数组范围的指针；
- 指向其他对象的不清晰指针；
- 凭想像和运气来使用的指针。

显然，我们不想使用这类指针。我们希望使用一直指向它们（我们，还有其他人）认为应该指向的对象的指针。对于程序员来说，这需要一些使用规则。所以，指针的第一条使用规则就是：

在迫不得已的情况下使用指针。

但只要有可能，就可以使用引用。你一定要避免使用未初始化的指针。使用引用也能让你较好地控制对象和变量的作用域。引用与指针不一样，它必须在定义之时初始化。如果它们位于一个对象内部，你必须在适当的位置定义此对象。必须在这些引用可用的作用域内创建它，这要遵循一定的创建顺序。

另一个指针的误用来源于 C，就是使用指针代替数组索引。人们以前认为，使用一个指针来扫描数组代码执行速度更快，但对于现代优化的编译器来说并非如此。事实证明，当算法用索引而不是用指针来表达时，编译器更能实现一些较大程度（且正确）的优化。所以，对于过去编程时代适合的这一准则现在已不适用了。我们将很快对此问题进行更详细的讨论。

最后，许多 C 程序员习惯于结合使用指针与错误代码，这就是说，使用一个空指针作为错误的符号。虽然也可以实现“空引用”，但由于显而易见的原因，本书不会讨论此问题（实在不值得推荐）。解决方案要么是将错误码与引用分离，要么是使用异常来标记不正常条件（参见第 10 章“资源管理”）。当然，设计返回指针的函数和使用空指针来标记特殊条件没有错。

对待指针，有一个持相反观点的学派——他们一直互相争论。一些面向对象的语言，尤其是 Java 和 C#，使用指针作为访问对象的唯一途径。它们不允许 C 样式的指针操纵。在 Java 中，指针被当成不变的句柄。而且，像 Java、C#、Smalltalk 和 Eiffel 这样的语言实现了垃圾回收机制。程序员不必再为跟踪指针及其所指的对象而费心了，因为运行时系统会自动管理这一切。它引用——计算每个对象并自动销毁没有任何指针再指向的对象。这一机制还不能随意获得，必须由运行时进行控制。而且，当执行析构函数时，由于不能直接控制而使得资源管理不可能在此系统中实现。但是，一切都将不断发展完善。我们将在后面的章节中进行更具体的讨论。

3.2 指针与引用

在类型名与变量名之间用一个星号来声明指针变量（星号与变量名结合使用）。例如：

```
int * pValue;
```

此语句声明 pValue 是一个指向整数的指针。指针可以用其他一些变量的地址（或通过赋值运算符=来赋值）进行初始化。运算符的地址由“&”标明（此&与引用中的&无冲突，它们出现在不同的场合）。例如：

```
int TheValue = 10;
pValue = &TheValue;
```

将变量 TheValue 的地址赋值给指针 pValue（如图 3-1 所示）。

如果我们想访问指针所指的值，我们要使用反引用运算符：星号*（再次说明，它的双重意义不会导致任何混淆）。例如：

```
int i = *pValue;
```

将 pValue 所指向的值赋值给整数 i。在此例中，我将此值改为 10。相反：

```
*pValue = 20;
```

更改了 pValue 所指值，也就是更改了 TheValue 的值（TheValue 现在的值为 20）。你可以看到，这比使用引用要稍微复杂一些。

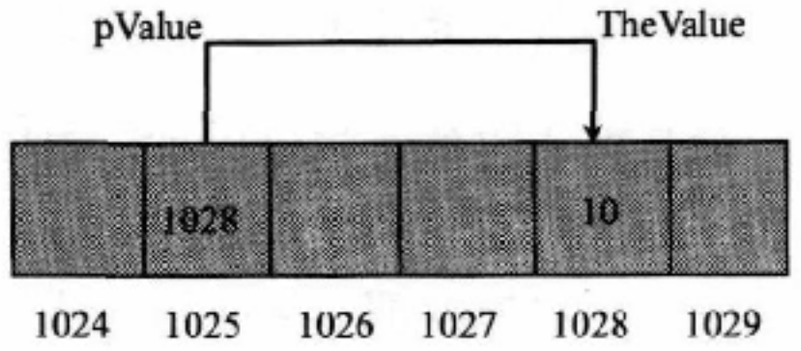


图 3-1　指针 pValue 保存变量 TheValue 的地址（即指针指向此地址）

顺便提及，同样的例子使用引用后则为：

```
int TheValue = 10;
int & aliasValue = TheValue;     //引用 TheValue
int i = aliasValue;              //通过一个别名访问 TheValue
aliasValue = 20;                 //通过别名改变 TheValue
```

3.3　指针与数组

通过指针访问数组、指针数组。

数组作为指针进行传递。事实上，C++数组以指向数组的第一个元素的指针形式来实现。在实际中，这意味着我们可以访问指针，仿佛它就是一个数组一样——即像使用一个索引一样。相反，我们可以使用指针来访问数组的元素。我们可以增加一个指针，从数组的一个元素移到另一个元素。当然，不管在哪种场合，我们都要确保指针实际指向数组的某个元素。

字符串是数组的典型例子。标准库中有一个函数叫 strlen，它用于计算以 null 结尾的字符串的长度（字符串在<cstring>头中声明）。让我们自己动手编写此函数的实现代码，我们称它为 StrLen。

```
int StrLen(char const str [])
{
  int i;
  for(i = 0; str [i] != '\0'; ++i)
     continue;
  return i;
}
```

这里没有使用空循环体（即只有一个分号的情况），而是使用了关键字 continue。它意味着忽略循环体的其余部分（如果有的话），并返回到循环头继续执行下一次迭代。

下面是给 StrLen 传递一个数组的 main 过程：

```
int main()
{
  char aString [] = "the long string";
  int len = StrLen(aString);
  std::cout << "The length of " << aString
            << " is " << len << std::endl;
}
```

这里要搜索终止 null，并返回此 null 的索引。非常明显，不是吗？

下面是一个更为传统的“优化”版本（如图 3-2 所示）。警告：此样式非常简洁，但不推荐采用！

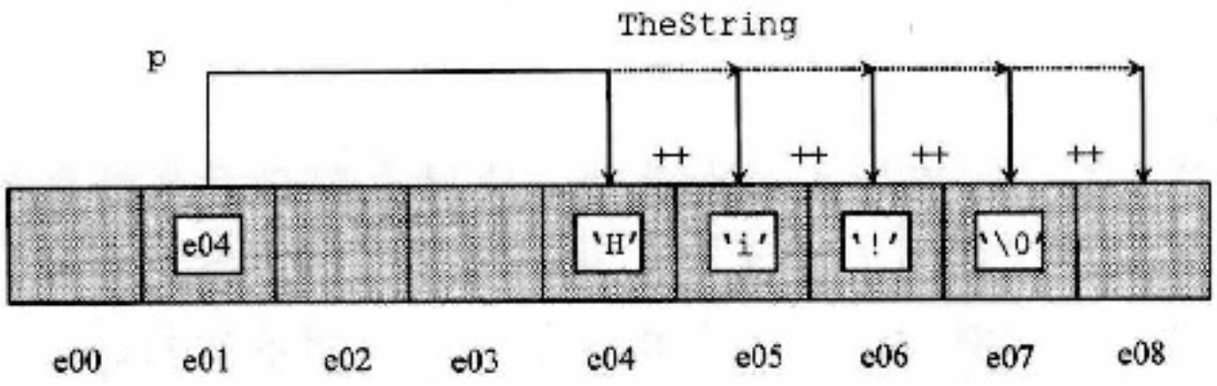


图 3-2 指针 p 初始指向字符串‘Hi!’地址 0xe04 处的第一个字符（使用十六进制符号表示地址）。后续的增加移动指针通过字符串的其余字符直到到达 null 字符

```
int StrLen(char const * pStr)
{
  char const * p = pStr;
  while(*p++)
    ;
  return p - pStr - 1;
}
```

将 p 初始化为指向字符串的起始处。从另一种角度讲，while 循环有点含糊。我们反引用指针，测试其布尔真值并执行后自增，一切都在一条语句内进行。如果通过反引用指针获得的字符不为 0（因为 0 等价于布尔值 false），循环将继续。

后自增运算符将指针移到数组的下一个位置，但只是当它在表达式中使用过（得到一个 true 或 false 值后）才如此。表 3-1 列出了所有的自增和自减运算符。

表 3-1 作用于变量 i 上的自增和自减运算符

	自增运算符（作用于 i）
i++	后自增
++i	前自增
	自减运算符（作用于 i）
i--	后自减
--i	前自减

本着简洁的原则，我在空循环体中没有使用 continue 语句——对于旧学派的 C 程序员来说，使用表示空语句的分号就足够了。

最后，我们将两个指针相减得到已经历的数组元素个数，再减去 1，因为我们还经过了

null 字符（最后测试访问 null 字符后才增加指针）。顺便说一下，我第一次也没有正确处理好此部分代码，导致我饱尝多次编辑——编译——运行之苦。

如果你理解第二部分实现代码有困难，这很正常。我不会劝说你不要编写这样的代码。但是，如果对比两种代码后，你仍然不愿意编写 StrLen 这样的较简单、可读性较强的索引实现过程，那就请你阅读下一节内容“谈谈汇编”。

编程艺术有着特殊的情形。它的发展如此快，使用早期 C 开始编程的人在此领域仍然非常活跃。在其他科学领域，这样的进步通常需要一代又一代的研究者努力才能达到。对于某一代看似新颖、革命性的思想对于下一代来说可能作为常识来接受。

另一方面，计算机的发展在短时间内翻天覆地，不可能让每个人都有时间来充分消化和接受所有的进步。仍然有许多程序员还在使用它们的古老编程技巧（虽然编程技术只有 20 年左右的历史），这还是他们在 64KB 内存 XT 计算机上编程时所学的技术。其中的少数一些人仍然拒绝接受新的现实，如果他们碰巧处于管理层，可能会带来较大的危害。

有一些优秀的旧书籍，它们是如此的优秀，以致于无人能超越它们。至今，新程序员还阅读如 Kernighan 和 Ritchie 的 *The C Programming Language* 之类的经典书籍来学习编程。它是一本好书，不要误会我的意思，但此书教授的是很久以前的编程方式。

算法和数据结构界的最高权威是 Donald Knuth 的经典著作 *The Art of Computer Programming*。它是一本精美且非常全面的科学著作。但是，我看见根据这些书籍中的算法编写的快速排序代码，它们是预构建编程的畸形。

最后，如果你的编译器不能优化可读性、算法的可维护版本，你必须兼任人工编译器——那就换一个新编译器吧!现在再没有人愿意充当作人工编译器。这也是为你自己和必须阅读此程序的编程同事着想。

在 C++中，可以按两种方式实现多维数组。如果你事先知道数组的维数，你可以使用多套括号来声明它。例如，一个 3×4 整数类型数组声明如下：

```
int arr2d [3][4];
```

而且可以通过指定两个索引来访问：

```
for(int i = 0; i < 3; ++i)
  for(int j = 0; j < 4; ++j)
    arr2d [i][j] = i * 10 + j;
```

另一方面，如果你不能确定行的长度，甚至让行的长度可变，最好使用一个指针数组。特别是，一套字符串可以作为相对字符的指针数组来实现。

下面是一个三个字符串数组的定义，通过初始化来完成：

```
char const * strings [] = { "one", "two", "three" };
```

这些字符串可以使用单个索引来访问：

```
char const * oneStr = strings [0];
```

而单个字符可以使用双索引来访问：

```
char cInitial = strings [0][0];
```

有趣的是，如果你希望程序通过命令行参数来调用，请按如下方式在 main 中进行声明：

```
int main(int argCount, char * arguments [])
```

当调用程序时，第一个参数 argCount 将包含保存在第二个参数数组 arguments 中的字符串数量。第一个字符串（确切地说为 0 字符串）arguments [0]将总是包含用于调用程序的名字。余下的字符串（从 arguments [1]到 arguments [argCount - 1]）将包含调用程序的命令行参数。例如，如果程序名为 testing.exe，你可以通过三个参数从命令行调用它：

```
testing one two three
```

通过 4 个字符串来调用 main（argCount 等于 4），参见表 3-2。

表 3-2 命令行参数

字 符 串	值
arguments [0]	"c:\work\testing.exe"
arguments [1]	"one"
arguments [2]	"two"
arguments [3]	"three"

3.3.1 谈谈汇编

“不美观，但人们关心的是，它是否更有效”，但事实上并非如此。对于那些理解 x86 汇编器的人，表 3-3 显示了两个实现代码的 16 位编译器优化输出。

在第一个实现代码中，编译器决定使用两个寄存器变量，以此来推入和弹出。循环本质上是一样的；只是寻址模式不同。经过仔细检查可以发现，第二个循环中的指令与第一个相比长 1 个字节。

80 39 00	cmp	BYTE PTR [bx][di],0	;第一个循环
80 7f ff 00	cmp	BYTE PTR [bx-1],0	;第二个循环

所以，对于实在太长的字符串，索引实现法就会优胜于指针法。为什么呢？大部分取决于指令队列。在旧式计算机上，如一台 80486，第二个循环证明排列得更好，所以代码运行更快。

在指针实现代码中，一些附加的指针算法在结尾处进行；在索引实现代码中，测试在进入循环前进行——然后才重复执行几次循环。同样在我的计算机上，索引方案比指针的开销要少，所以，对于多达 3 个字符的字符串，索引方案胜过了指针方案。

表 3-3　　对于 StrLen 的两个实现代码，比较由某优化编译器生成的汇编代码

索　引	指　针
?StrLen@@YAHPAD@Z PROC NEAR	?StrLen@@YAHPAD@Z PROC NEAR
push bp	push bp
mov bp,sp	mov bp,sp
push di	
; pStr=4	; register bx=p
; register bx=i	; pStr=4
mov di,WORD PTR [bp+4]	mov dx,WORD PTR [bp+4]
xor bx,bx	mov bx,dx
cmp BYTE PTR [di],bl	
je $FB1596	
$F1594:	$FC1603:
inc bx	inc bx
cmp BYTE PTR [bx][di],0	cmp BYTE PTR [bx-1],0
jne $F1594	jne $FC1603
$FB1596:	
mov ax,bx	mov ax,bx
pop di	**sub ax,dx**
	dec ax
mov sp,bp	mov sp,bp
pop bp	pop bp
ret	ret
?StrLen@@YAHPAD@Z ENDP	?StrLen@@YAHPAD@Z ENDP

坦白地说，两者各有千秋。它值得复杂化吗？你确实比较过所有常用技术的汇编指令与时效性吗？也许现在可以抛弃来自 C 的陈规陋习，开始学习一些新的技巧，着重提高代码的可理解性和可维护性。我想，你并不想因小失大。

不要在可以使用索引的地方使用指针。

3.3.2　练习

1. 实现下面函数

```
void StrCpy(char * strDest, char const * strSrc);
```

它复制源字符串到目标字符，包括终止 null。假定目标字符有足够的空间（但实际是我

们不能简单地如此断言)。使用“索引数组”方法而不用 char 的指针。

2. 实现下面函数

```
int StrCmp(char const * str1, char const * str2);
```

按词典方式比较两个字符串。当两个字符串相等时，返回 0；当第二个字符串“小于”第一个字符串时返回一个正数；当第二个字符串“大于”第一个字符串时返回负数。

字符串逐个字符地进行比较，直到找到不匹配字符为止。如果 str1 中第一个不同的字符 ASCII 码大于 str2 中的对应字符，则返回值为正数；否则，返回负数。当在某一个字符串中碰到 null 字符时终止比较。如果 str1 较长，则返回一个正数；如果 str2 较长，返回一个负数。使用索引来实现。

3. 另一个古老的“优化”技巧是逆向运行循环。此时不是增加计数器，而是减小计数器直到它变成一个负数或 0（当循环变量为无符号类型时，运用此技巧的每个程序员都会遇到不同程度的挫折)。

找出下面函数的漏洞并重新编写：

```
void StrNCpy(char * strDest,
             char const * strSrc, std::size_t len)
{
  while(len--)
    *strDest++ = *strSrc++;
}
```

其中，std::size_t 为一些无符号整数类型，例如在 cstring 中。

通过人工编译器舍弃这里介绍的“优化”技术。

4. 编写一段测试程序来打印它的命令行参数。

3.4 指针与动态内存分配

尽管本书介绍了指针的一些负面作用，但指针用于动态数据结构时，它的作用是非常大的。动态数据结构必须能伸缩。“伸”包括对新内存空间的需求和维护，“缩”意味着回收它们。我们如何获得新内存空间呢？我们必须请求系统为特定类型的变量或对象分配内存。使用运算符 new 来完成这一点。由 new 返回的内存从所谓的堆（heap)，即由 C++运行时管理的内存区域开始分配。如何访问此新内存呢？使用一个此特定类型变量或对象的指针来实现。例如，为了获得足够的内存来保存单个整数[1]，我们使用如下语法：

```
int * pNumber = new int;
```

更合理的是，我们可以（也必须）用一个需要保存于此的特定值立即初始化此内存（如图 3-3 所示)。

[1] 如果整数的生存期与某个方便的作用域对应，你可以将此整数保存为一个本地变量。但如果你要保存整数的生存期不易简单限制于某个作用域，或者不能事先知道所需的整数个数，你就不能对它们都使用本地变量，唯一的选择就是动态分配空间。

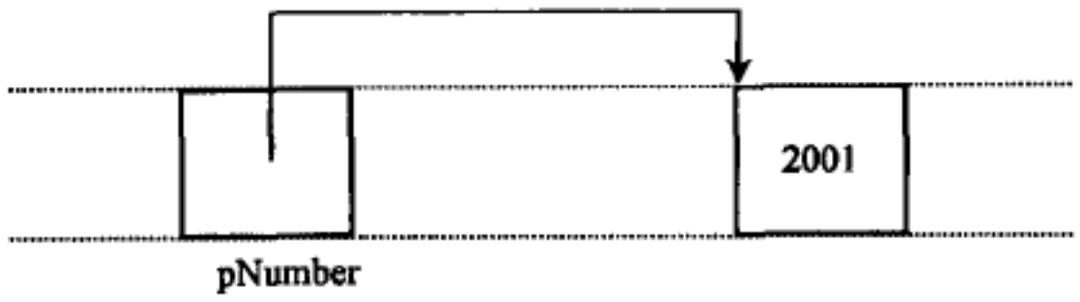


图 3-3　为一个整数分配新内存并初始化为值 2001。指针 pNumber 包含这一新内存的地址

```
int * pNumber = new int(2001);
```

当分配一个带有构造函数的对象时，初始化是强制性的。有两种情况：第一种情况，当构造函数不带任何参数，你不必做任何事——此构造函数按缺省情况调用。例如，缺参数构造函数可能初始化成员变量为一些预定义值（如 0）。

```
IStack* pStack = new IStack; // 调用缺省构造函数
```

第二种情况是，当对象有一个需要参数的构造函数时，我们必须在调用 new 时指定它的值：

```
Star * pStar = new Star(1234.5, 10.2);
```

如果我们想保存 n 个整数，需要分配 n 个整数的数组（如图 3-4 所示）。

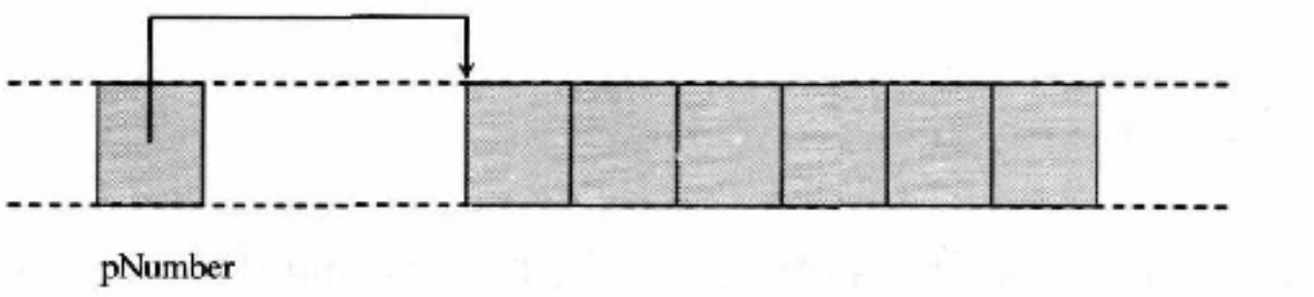


图 3-4　分配数组的结果

```
int * pNumbers = new int [n];
```

同样，可以分配一个没有构造函数的对象数组，例如：

```
Calculator * calcs = new Calculator [100];
```

请注意，即使一个对象没有显式构造函数，编译器也可能不得不生成一些代码来构建它。例如，Calculator 中嵌入了 IStack，因此，Calculator 缺省构造函数必须调用 IStack 的（缺参数）构造函数。你不必编写（声明或定义）此构造函数——编译器自动完成这一切。当需要初始化嵌入对象或基类时，编译器总是生成一个不可见的构造函数（以及析构函数）。这意味着上面的代码实际上将执行 IStack 构造函数 100 次——每一次针对一个 Calculator。

当然，如果一个类有一个显式定义的无参数构造函数，则此构造函数将由编译器在每次分配时调用，如下所示：

```
IStack * stacks = new IStack [100]
```

但是，如果我们想动态分配一个有非无效构造函数的对象数组时，我们将如何做呢？例如，带有一个 CelestialBody。它有一个构造函数，此构造函数需要一个参数——质量值。如果编写下面代码：

```
CelestialBody * bodies = new CelestialBody [100];
```

编译器将拒绝编译它。一旦某个对象具有非无效构造函数，编译器（正确地）假定它不能继续执行，它将产生一个缺省的无参数构造函数，例如：

```
CelestialBody(): _mass(0){}
```

你可以在类定义中保持两种构造函数，只要它们的参数类型或数量不同。一般而言，对多个方法使用相同的名字就称做重载（overloading）。你必须确保编译器根据调用时所传递的参数类型和数量就能清楚知道该调用哪一个重载方法。

最后，你可以为参数提供一个缺省值来结合 CelestialBody 的两个重载构造函数。在类定义中，你可以写为：

```
CelestialBody(double mass = 0.0): _mass(mass){}
```

这等价于具备参数个数为 0 和 1 的两个重载构造函数。应用它可以写出如下代码：

```
CelestialBody * bodies = new CelestialBody [100];
```

你可以为任何一个方法的参数提供缺省值。唯一的限制就是缺省参数必须跟随非缺省参数。当调用此方法时，可以跳过其后的所有参数，编译器将使用方法（或构造函数）定义中已经指定的缺省参数。

没有直接初始化动态分配数组内容的途径，因为没有办法给对象的构造函数传递初始值或参数。我们必须不断遍历新分配数组并手工设置其值。例如：

```
for(int i = 0; i < n; ++i)
  pNumbers [i] = i * i;
```

为了回收或释放通过 new 分配的内存，我们使用运算符 delete：

```
delete pNumber;
```

如果被删除的对象有一个构造函数，它将自动被调用。例如：

```
delete pStar;
```

将打印“Destroying a star of brightness……”

在数组中，在欲删除的指针前面放一对方括号（我们将括号放在指针名字的前面而不是放在它的后面）。

```
delete [] pNumbers;
```

再次说明，如果我们处理的是对象的数组，则每个对象的析构函数将被自动调用。

在指针的数组中并非如此！当删除指针的数组时，数组元素所指向的对象不会自动被删除。你必须通过实在的循环来删除它们。一般的经验是：尽量在某个对象的构造函数中分配内存，在析构函数中解除分配。分配的解除应该反映分配。如果必须在循环中调用运算符 new，你必须在同样的循环中调用运算符 delete。

删除一个空指针是安全的。提示：

在调用 delete 前不要检查空指针。

delete 将为你完成一切。

如果我们对一个动态分配对象或数组忘记了调用 delete，将会发生什么呢？它所占用的内存永远也不会被回收（至少在程序结束前不会）。我们称之为内存泄漏。这是一个 bug 吗？在程序终止时，所有的内存都被释放了（虽然没有调用泄漏对象的析构函数），那么，什么人会关心这样的泄漏呢？

具体情况具体分析。如果在程序执行期间，重复泄漏了越来越多的内存，它将最终耗尽内存。因此，这是一个 bug（而且此错误难于发现）。因此，良好的编程习惯是，总是释放任何分配过的一切资源。顺便提及，有一些工具和技术可用于监测内存泄漏。我将在后面讨论它。

3.5　动态数据结构

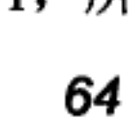
链表、递归类型定义、哈希表、数组与指针。

现在是介绍几个基本数据结构的时候了，这些结构是创建更多复杂对象的基石。我们将从一个动态数组（实际上是一个堆栈）、链表及字符缓冲器开始讲解，然后介绍哈希表，最后综合运用这些组件形成一个字符串表——有效保存和读取字符串的数据结构。

在 C++标准库中可以找到几乎所有的数据结构，知道这一点对编程有好处。一个动态数组称做一个矢量；其中也有一个列表、字符串等。不过，我首先介绍如何实现这些数据结构而不应用标准库有以下几个原因。首先，这是一种一步一步地学习和练习语言的好方式。其次，我希望读者对于这些数据结构是如何在标准库中实现的有一个了解。最后，标准库使用了非常高级的语言特征（例如模板、运算符重载等），而这些概念我们尚未学习。在以后的章节中我们将会学习到。

3.5.1　动态堆栈

动态数组。



calc4

我们现在将修改堆栈的契约。对推入的整数数量不再有严格的限制，我们可以在计算机内存耗尽前一直推入数据。

我们不愿意让程序成为一个内存独占者（至少在没有必要时不是这样），因此，我们不想预分配最大的可用内存块。相反，我们将分配一些最小的字节数，而且每次用完堆栈空间时就加倍它的大小。显然，我们不再需要方法 IsFull。除此之外，接口与原来的堆栈相同。

新的堆栈工作方式为：有一个私有帮助函数 Grow，它用于在内部加倍堆栈大小。将堆栈的当前容量保存在私有数据成员_capacity 中。为了测试目的，我们将堆栈初始容量设置为 1，所以，我们几乎可以立即看到堆栈的容量增大。

```
const int initStack = 1;
class Istack
{
public:
  IStack();
  ~IStack();
  void Push(int i);
  int Pop();
  int Top()const;
  bool IsEmpty()const;
private:
  void Grow();
  int * _arr;
  int   _capacity; // 数组大小
  int   _top;
};
```

注意其中的_arr 声明为一个指针而不是一个数组。一个动态分配的数组必须声明为一个指针。

在 IStack 的构造函数中分配初始数组，并且在析构函数中删除内存。

```
IStack::IStack()
  : _top(0), _capacity(initStack)
{
  _arr = new int [initStack]; // 分配内存
}

IStack::~IStack()
{
  delete [ ]_arr; // 释放内存
}
```

以上代码示例了对象间的另一种新关系。我们不仅可以说，IStack 可以访问此数组，还可以说 IStack 拥有此数组（如图 3-5 所示）。使用一个指针来表达 owns-a 的关系。如果对象 A 负责对象 B 的存储单元释放，对象 A 就拥有对象 B。当我们讨论资源管理时，就会更详细地阐述这一点。

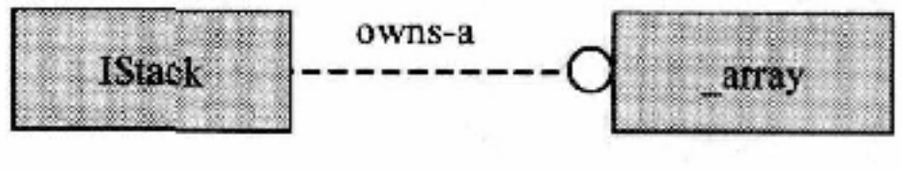


图 3-5 对象间的 owns-a 关系

IStack 在构造函数中请求数组的内存，在析构函数中释放内存。待一会儿，我们将看到方法 Grow 对自身资源的重新分配。

当我们推入一个整数但数组中无剩余空间时，会发生什么情况呢？我们要调用 Grow 方法并完成 Push。

```
void IStack::Push(int i)
{
  assert(_top <= _capacity);
  if(_top == _capacity)
```

```
    Grow();
  _arr [_top] = i;
  ++_top;
}
```

让我们看一下 Grow 的“所作所为”。它必须经过以下几步：

1. 分配一个两倍于当前数组大小的新数组。
2. 将旧数组中所有的数据项复制到新数组的第一个二分之一部分。
3. 加倍_capacity 变量。
4. 删除旧数组。
5. 设置指针_arr 指向新分配的数组。

```
void IStack::Grow()
{
  std::cout << "Doubling stack from "
            << _capacity << ".\n";
  // 分配新数组
  int * arrNew = new int [2 * _capacity];
  // 复制所有数据项
  for(int i = 0; i < _capacity; ++i)
    arrNew [i] = _arr [i];
  _capacity = 2 * _capacity;
  // 释放旧内存
  delete []_arr;
  // 用新数组替换旧数组
  _arr = arrNew;
}
```

语句

```
_arr = arrNew;
```

是一条指针赋值语句。我们让指针_arr 改成指向新位置——即 arrNew 所指向的同一个位置。这一操作是运用引用不能实现的。请注意，我们不能更改此位置的内容。这由以下语句完成：

```
*_arr = *arrNew;
```

或其等价语句：

```
_arr [0] = arrNew [0];
```

我们更改了指针_arr 本身。

```
_arr = arrNew;
```

为了完整性，下面提供了余下方法的实现代码。请注意，虽然_arr 是一个指针，但对它可像一个数组一样进行访问。

```
int IStack::Pop()
{
  //不要对空堆栈执行 Pop
  assert(_top > 0);
```

```
  --_top;
  return _arr [_top];
}
int IStack::Top()const
{
  //不要对空堆栈调用 Top
  assert(_top > 0);
  return _arr [_top - 1];
}
bool IStack::IsEmpty()const
{
  assert(_top >= 0);
  return _top == 0;
}
```

表达式_top == 0 的所得值为布尔值 true 或 false，而且此结果从 IsEmpty 返回。我们现在可以在 calculator 中换成此 IStack 的新实现代码并测试它了。注意，我们从未压缩过此堆栈。这称为高水位线类型算法。不过，也容易添加这一 Shrink 方法，它在_top 每次小于_size/2 时被 Pop 调用。

3.5.2 链表

链表是一种针对快速定时（独立于列表大小的）、预先挂起及线性时间（与列表大小成正比）检索的动态数据结构。线性时间检索意味着随着列表的增大，检索时间也变长。列表中包含链接。假如我们想在列表中保存一些整数 ID，链接将包含 ID 的字段及指向下一个链接的指针。为了查看它的实际工作方式，让我们看一下图 3-6。

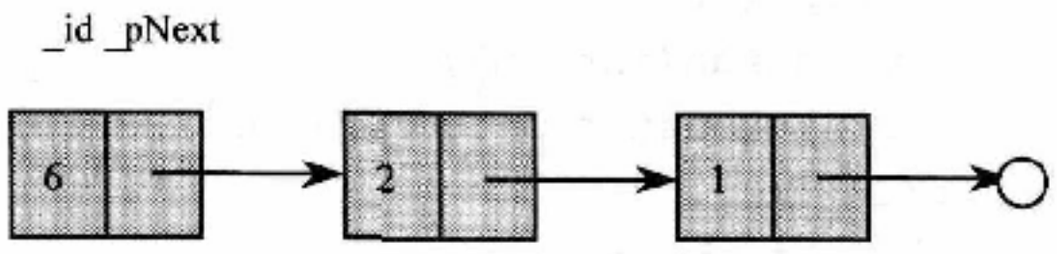


图 3-6 保存数字 6、2 和 1 的链表

列表中的最后一个链接包含 null（0）指针_pNext，这是列表的终止符。一个 null 指针永远不会是一个有效指针，也就是说，它不能反引用。但是，它可检查是否为 0。

list

下面是类 Link:

```
class Link
{
public:
  Link(Link* pNext, int id)
    : _pNext(pNext), _id(id){}
  Link * Next()const { return _pNext; }
  int   Id()const { return _id; }
private:
```

```
  int   _id;
  Link *_pNext;
};
```

注意，Link 的定义为自引用——一个 Link 包含一个指向 Link 的指针。对编译器不知道定义的类型使用指针和引用是可行的。在这种情况下，编译器惟一所需要的就是类型的声明。在这里是第一行：

```
class Link
```

正作为这样的一个声明（不要与前面的声明混淆——参见 5.4 节的内容）。

链表保存指向第一个链接的指针。从概念上讲，列表是一个与链接不同的对象。首先，它们的接口不一样。程序员可以给列表添加新的项目，还可以删除它。程序员也可以重复遍历列表。这些操作对于 Link 的方法没有意义。然而，从某种程序上讲，每个链接都从一个列表开始。这种考察事物的递归方式在 LISP 之类的语言中很普遍。我们在面向对象编程中通常尽量避免此类递归概念。所以，对于我们来说，List 与 Link 并不一样。

从_pHead 指向的第一个链接开始，我们可以跟随指针_pNext 遍历整个列表。

在起始时，列表为空而且指针初始化为 0。

```
class List
{
public:
  List(): _pHead(0){}
  ~List();
  void Add(int id);
  Link const * GetHead()const { return _pHead; }
private:
  Link* _pHead;
};
```

通过分配一个新链接并且用 ID 和指向下一个项目的指针初始化给列表添加一个新的 ID。下一个项目为先前位于列表起始处的链接（_pHead 指向它）。新链接变成了列表的头，而且_pHead 指向它（指针赋值），如图 3-7 所示。此过程称做预先挂起（prepending）。

```
void List::Add(int id)
{
  // 添加在列表头
  Link * pLink = new Link(_pHead, id);
  _pHead = pLink;
}
```

因为列表动态增长，所以，每个新链接必须使用运算符 new 来分配。当分配一个新对象时，它的构造函数被自动调用。这就是为什么我们在 new Link（_pHead, id）中传递构造函数参数的原因。

列表迭代，即逐个元素经过整个列表，此过程非常简单。例如，下面是我们检索一个包

含给定 ID 链接的方式：

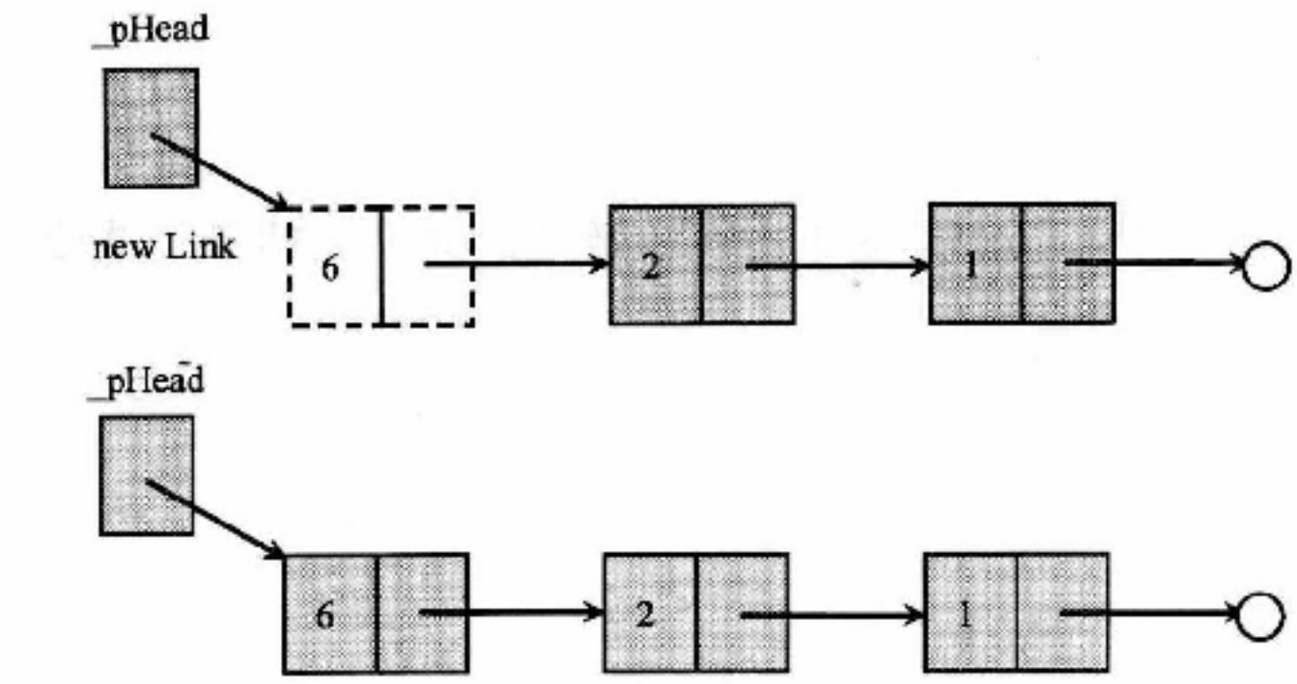


图 3-7　在列表前预先挂起一个新元素

```
for(Link const * pLink = list.GetHead();
    pLink != 0;
    pLink = pLink->Next())
{
  if(pLink->Id()== id)
     break;
}
```

在 pLink->Next()中，我们调用了通过指针访问过的某个对象的成员函数。我们必须反引用指针，然后访问成员函数。这可以显式完成，如（*pLink）.Next()，或者使用简写法 operator ->，它组合了两个动作（反引用以及成员访问）。

任何类型的循环都可以使用 break 语句立即退出循环体。相对于相同代码的简洁版本：

```
for(Link const * pLink = list.GetHead();
    pLink != 0 && pLink->Id()!= id;
    pLink = pLink->Next())
{
  continue; // 空循环体
}
```

我更喜欢使用前一种样式（虽然它没有严格地遵守结构化编程的原则）。

以上代码在循环头中混合了两个退出条件。事实上（而且这也会受到结构化编程拥护者的攻击），你可以使用一个 return 语句来从循环中退出。例如，下面是某程序员实现 Find 函数的代码：

```
bool List::Find(int id)const
{
  for(Link const * pLink = GetHead();
      pLink != 0;
      pLink = pLink->Next())
  {
```

```
    if(pLink->Id()== id)
        return true;
  }
  return false;
}
```

此列表有一个析构函数，其中释放了所有的链接。此列表拥有此链接。

```
List::~List()
{
  // 释放列表
  while(_pHead != 0)
  {
    Link* pLinkTmp = _pHead;
    _pHead = _pHead->Next(); // 取消链接 pLink
    delete pLinkTmp;
  }
}
```

释放列表的算法工作方式如下：只要列表不为空，我们就分配一个临时指针 pLinkTmp，让它指向第一个链接；让_pHead 指向第二个链接（可能为空），这等价于与第一个链接解除链接，而且删除此链接（如图 3-8 所示）。语句

```
_pHead = _pHead->Next();
```

是一个指针赋值。_pHead 现在指向由_pHead->_pNext 所指的同一个链接。

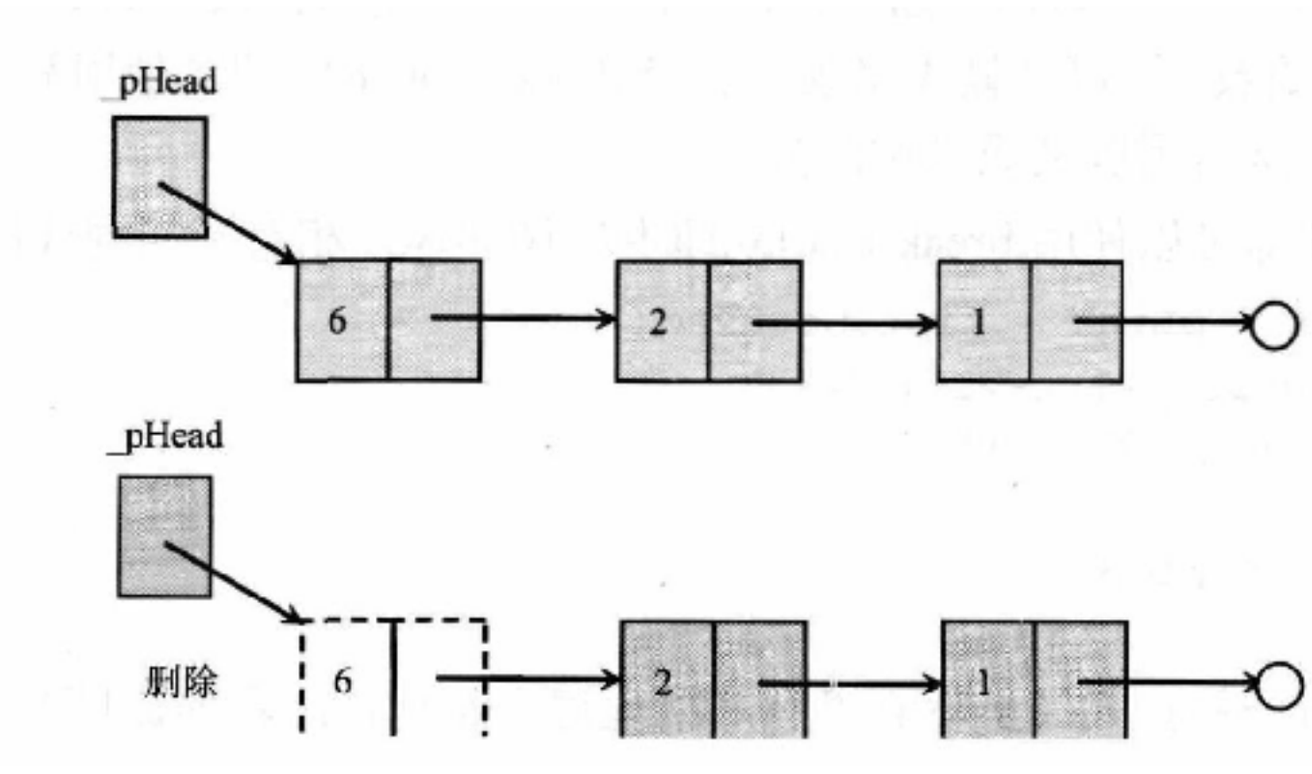


图 3-8　取消链接和删除列表的第一个元素

再次说明，有一些事是不能用引用完成的。

有一种链表析构函数的更简单、递归性实现方式：

```
List::~List()
{
  delete _pHead;
}
Link::~Link()
```

```
{
  delete _pNext;
}
```

当销毁一个 Link 时，它删除了它的后续者，后续者反过来又会删除它的后续者，如此等等。当它到达终止空指针时，递归会自动停止，因为对于一个空指针而言没有任何析构函数可调用。

递归解决方案的明显优点是其简单性。为此简单付出的是速度的减慢和内存的开销。递归函数调用通常比循环开销更大（尽管对所谓的尾随递归可进行编译器优化，将递归转换成迭代）。递归函数调用还消耗了程序的堆栈空间。如果堆栈空间非常珍贵（例如编写操作系统内核时），就不可能使用递归。但是，即使你有充足的堆栈，你可能仍然要选择迭代方案来处理非常大的链表。另一方面，因为内存的消耗，应用非常大的链表也是下策。在任何情况下，不管采取哪种途径，都必须把握好平衡点。

同时要注意代码中指向常量指针的用法。方法 GetHead 声明为返回一个指向常量 Link 的指针。

```
Link const * GetHead()const;
```

所以，我们赋值的变量必须也是一个指向常量 Link 的指针。

```
Link const * pLink = list.GetHead();
```

指向常量指针的意义与引用一个常量的意义相同——所指向的对象，或引用的对象不能通过此类指针和引用进行更改。但是，对于指针来说，我们必须区分指向常量的指针与常量指针。后者是一个指针，一旦初始化，就不能指向其他任何对象（与引用一样）。这两种场合的语法如下：

```
Link const * pLink;                    // 指向常量的指针
Link * const pLink = pInitPtr;         // 常量指针
```

最后，可以将两者组合成一个指向常量的常量指针：

```
Link const * const pLink = pInitPtr;   // 指向常量的常量指针 ptr
```

最后一个指针仅能用于读访问被初始化为单个的位置。

当提到指针、引用与常量的组合时，会产生一些混淆。因为对于引用只有一种为常量的可能性，程序员“对常量的引用”与“常量引用”通常交替使用术语。但是，这两种情况相对于应用指针的场合时常常会导致混淆。在本书中，常量指针总是指一个不能移动的指针，而指向常量的指针表示不能通过它进行写操作。

稍等一下，还有让你混淆的地方！对于指向常量的指针有一个等价语法：

```
const Link * pLink;                    // 指向常量的指针
```

而且当然有：

```
const Link * const pLink = pInitPtr;   // 指向常量的常量指针 ptr
```

产生此混淆的根源在于我们从左至右阅读文本的固有习惯（中东地区的人会嘲笑我们）。因为 C 是为计算机而不是为人编写的，阅读的方向关系不大，所以，现在最好从右至左阅读声明。

让我们反向一些声明。记住星号的意思是“指向一个”。下面的声明：

```
Link const * pLink;
```

就变成了图 3-9 所示的样子。

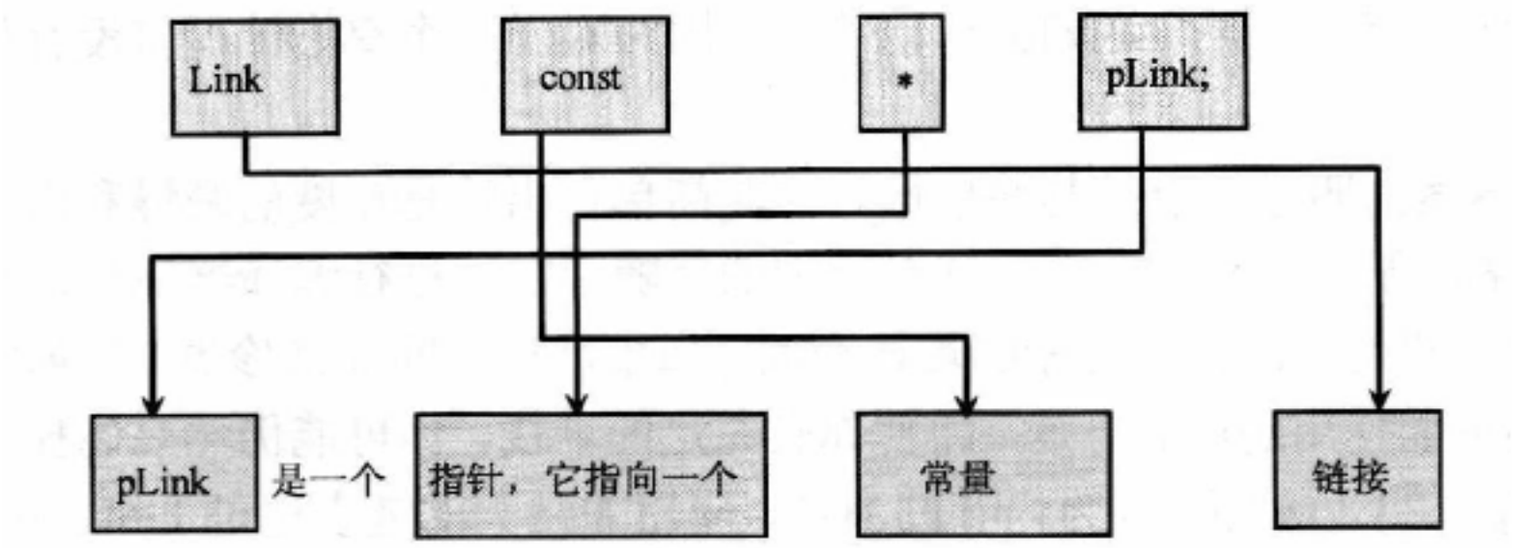


图 3-9　读类型声明：pLink 是一个指向常量 Link 的指针

同样

```
Link * const pLink;
```

变成了图 3-10 所示的样子。

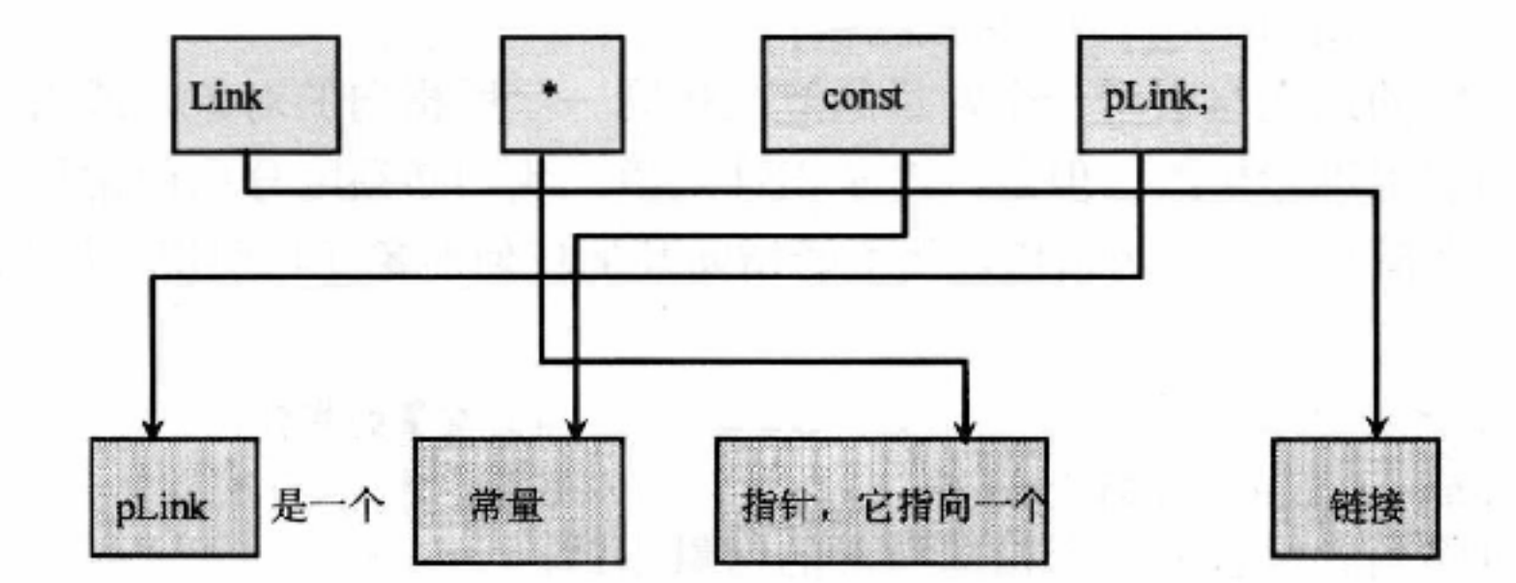


图 3-10　读类型声明：pLink 是一个指向 Link 的常量指针

下面的代码

```
Link const * const pLink = pInitPtr;
```

变成了图 3-11 所示的样子。

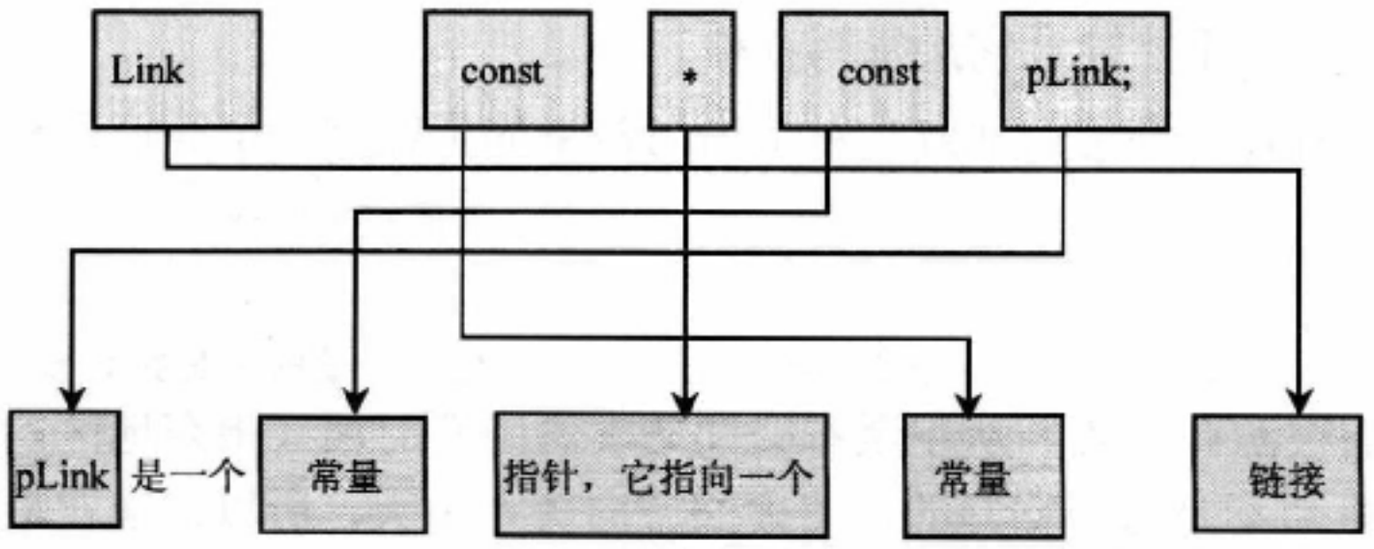


图 3-11　读类型声明：pLink 是一个指向常量 Link 的常量指针

3.5.3 字符串表

字符串表是整数到字符串、字符串到整数的映射。第一个映射使用一个字符串（指针，或用更理想的缓冲器偏移）的数组容易实现。第二个映射更难有效实现。

我们的目标是拥有定时访问的两种途径，即意味着，从一个整数到一个字符串的映射所花时间大约相同，与表中有多少字符串无关（在合理的范围内）。按程序员的话讲，就是 big-O of 1 或 O（1），或者是恒定时间。相反的映射（从字符串到整数）也如此。请注意，在最简单的实现过程中（遍历字符串列表并与检索字符串比较），执行所花时间随列表的增长而不断增长（我们说它呈线性增长或 O（N））。不过，有一种称做哈希表的数据结构可以完成定时映射。我们将使用它来实现一个有效的字符串表。



strtab

为了实现这两个目标，我们准备开始设计此字符串表。因为我们希望能够添加许多字符串，所以，我们需要一个大的缓冲器来保存它们。StringBuffer 对象保存一个字符串并返回一个可以发现此字符串的偏移量。反过来，此偏移量保存在另一个数据结构偏移量数组中，它是整数 ID 到字符串偏移的映射。每个字符串因此赋值一个 ID，它是偏移量数组的索引（如图 3-12 所示）。

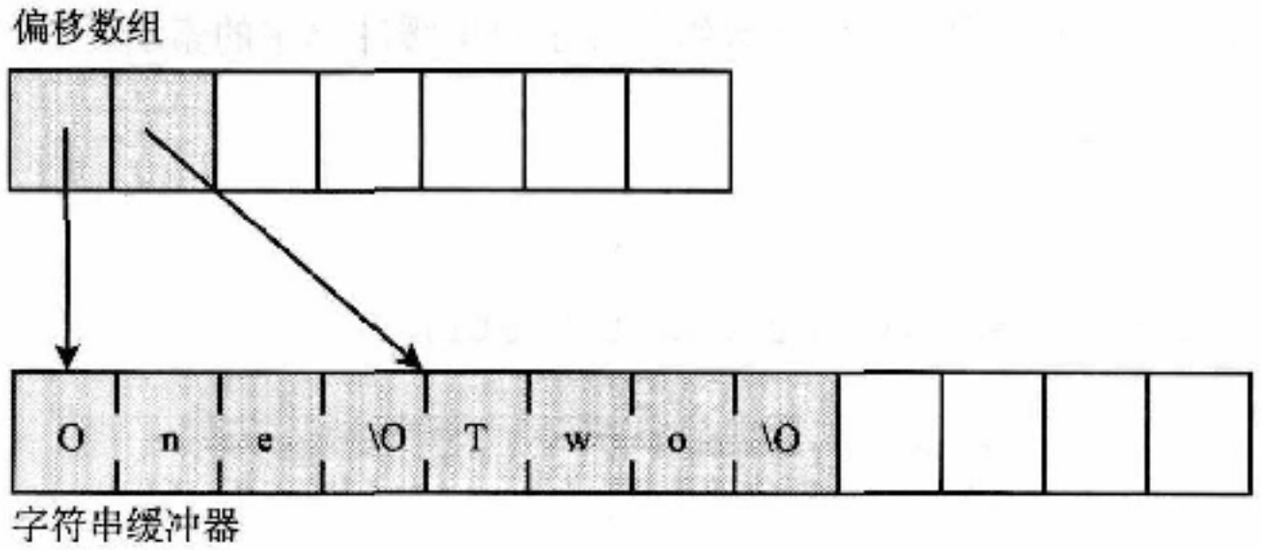


图 3-12 字符串缓冲器的偏移保存在一个数组中。此数组的索引用作字符串 ID

当添加一个新字符时，对应的条目也被添加到哈希表。暂时，让我们将哈希表看作一个映射字符串到 ID 的黑匣子。

```
const int idNotFound = -1;
const int maxStrings = 100;
//映射字符串到整数及整数到字符串的字符串表
class StringTable
{
public:
  StringTable();
  int ForceAdd(char const * str);
  int Find(char const * str)const;
  char const * GetString(int id)const;
```

```
private:
  HTable        _htab; // 字符串 -> IDs
  int           _offStr [maxStrings]; // ID -> 偏移
  int           _curId;
  StringBuffer  _strBuf; // 偏移 -> 字符串
};
```

从一个 ID 翻译成一个字符串由 GetString 方法完成，从字符串翻译到 ID 由 Find 完成，不查找相同项目的新字符串添加由 ForceAdd 完成。字符串表的设计如图 3-13 所示。

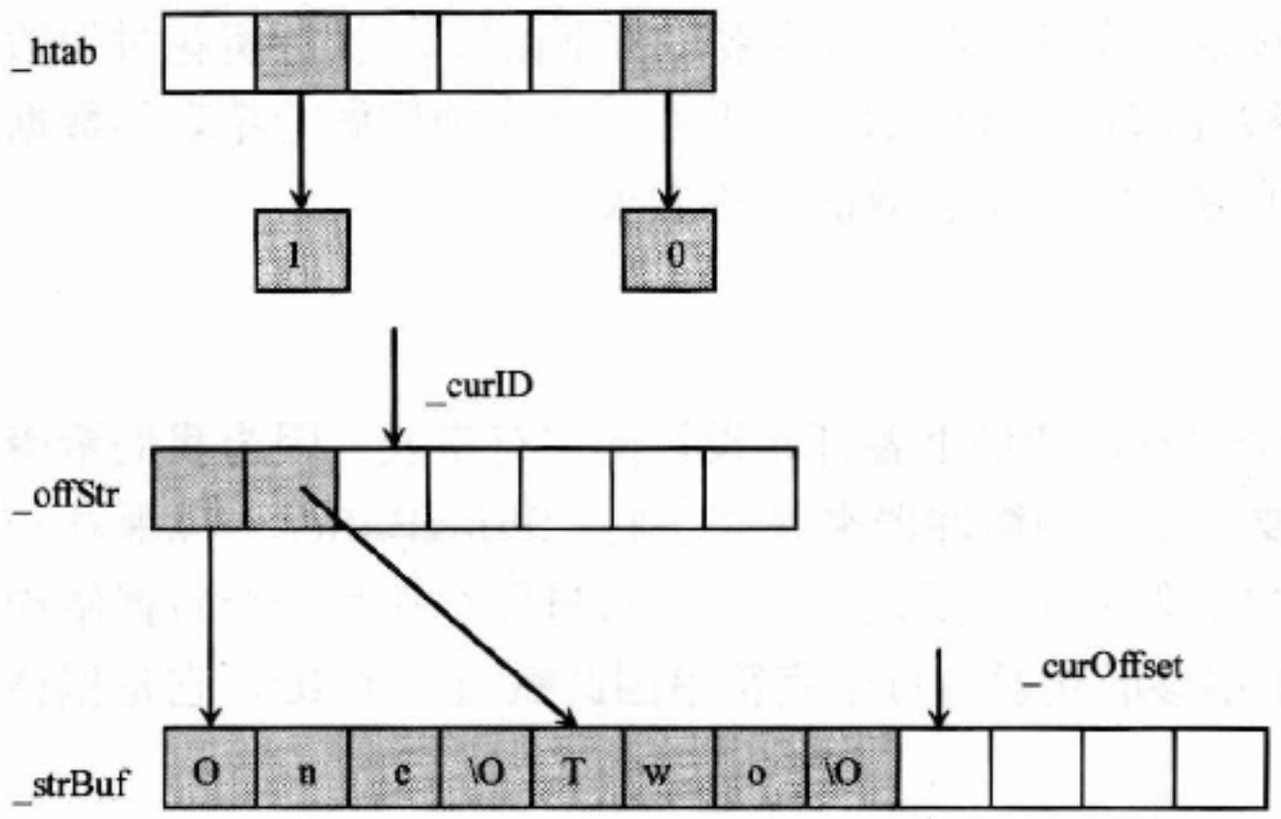


图 3-13　哈希表保存偏移数组中的索引，偏移数组保存字符串缓冲器中的索引，字符串缓冲器保存字符串

```
StringTable::StringTable()
  : _curId(0)
{}
int StringTable::ForceAdd(char const * str)
{
  int len = std::strlen(str);
  // 有足够空间吗?
  if(_curId == maxStrings || !_strBuf.WillFit(len))
  {
     return idNotFound;
  }
  // 指向字符串即将保存的地方
  _offStr [_curId] = _strBuf.GetOffset();
  _strBuf.Add(str);
  // 给哈希表添加映射
  _htab. Add(str, _curId);
  ++_curId;
  return _curId - 1;
}
```

哈希表（对我们来说仍然是一个黑匣子）查找短条目列表，其中之一包含有给定字符串的 ID。为了找到正确的 ID 而检索此列表。

```
int StringTable::Find(char const * str)const
{
  // 从哈希表获得短列表
  List const & list = _htab. Find(str);
  //重复此列表
  for(Link const * pLink = list.GetHead();
      pLink != 0;
      pLink = pLink->Next())
  {
    int id = pLink->Id();
    int offStr = _offStr [id];
    if(strBuf.IsEqual(offStr, str))
      return id;
  }
  return idNotFound;
}
// 映射整数到字符串。必须为有效 ID。
char const * StringTable::GetString(int id)const
{
  assert(id >= 0);
  assert(id < _curId);
  int offStr = _offStr [id];
  return _strBuf.GetString(offStr);
}
```

3.5.4 字符串缓冲器

字符串缓存作为一个字符数组来实现。我们一直复制 null 终止字符串到此数组，直到用完所有空间为止。变量_curOffset 是此数组的一个索引，而且它总是指向下一个可以复制字符串的自由空间。因为它被初始化为 0，所以指向缓冲器的开始。

在添加新字符串之前，我们确保它能放到余下空间中。添加一个字符串意味着将它复制到_curOffset 指向的地方，而且_curOffset 越过字符串的终止 null（如图 3-14 所示）。

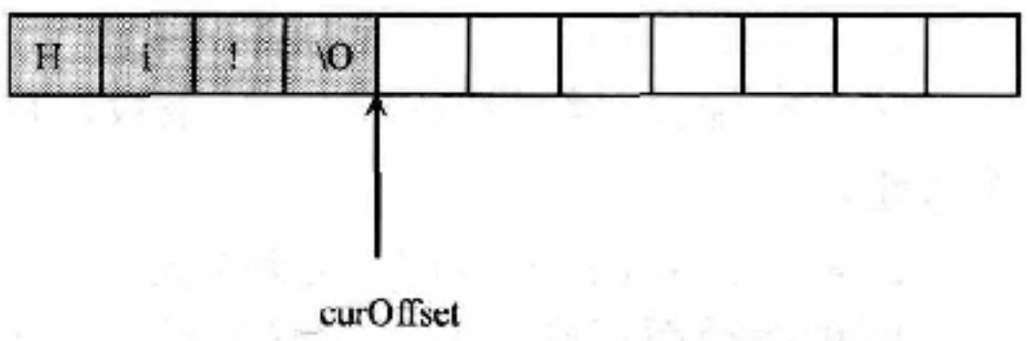


图 3-14 字符串缓冲器中的当前偏移

GetOffset 返回当前偏移，它可在后面用于访问即将复制的字符串。IsEqual 比较给定偏移的字符串与已知的字符串，GetString 返回给定偏移的常量字符串。

```
const int maxBufSize=500;
class StringBuffer
{
public:
  StringBuffer(): _curOffset(0){}
  bool WillFit(int len)const
```

```
{
  return _curOffset + len + 1 < maxBufSize;
}
void Add(char const * str)
{
  // 与 std::strcpy(_buf + _curOffset, str);相同
  std::strcpy(&_buf [_curOffset], str);
  _curOffset += std::strlen(str)+ 1;
}
int GetOffset()const
{
  return _curOffset;
}
bool IsEqual(int offStr, char const * str)const
{
  // char const * strStored = _buf + offStr;
  char const * strStored = &_buf [offStr];
  // 当字符串相同时，strcmp 返回 0
  return std::strcmp(str, strStored)== 0;
}
char const * GetString(int offStr)const
  {
    //与 return _buf + offStr;相同
    return &_buf [offStr];
  }
private:
  char  _buf [maxBufSize];
  int   _curOffset;
};
```

有两种途径创建一个指向数组中特定项的指针。一种途径是带上第 n 个元素的地址：

```
p = &arr [n]; //第 n 个元素的地址
```

另一种途径是给指针添加一个整数，目的是将它前移 n 项（注意，不是 n 个字节而是 n 个数组项）。

```
p = arr + n; // 给指针添加整数
```

两种途径都正确，而且生成了相同的代码——在上面的例子中，指针等价意义标明在其注释中。数组标记似乎不太容易混淆，所以，我们将继续使用这一方式。

我还使用了一般的赋值运算符+=，它将右边的表达式加到左边的变量上。例如：

```
n += i;
```

等价于：

```
n = n + i;
```

当然，生成的代码相同（尽管最初+=运算符由 C 作为一种优化而引入），而且当你习惯后，这种简写法就会变得有用。

最后，我使用了一些标准库中的字符串函数（更准确地说，C 和 C++中共有的标准库部分）。包含声明的头文件称做 cstring。在编译器帮助中可以找到这些函数的更详细描述。

3.5.5 表查找

在解释哈希表如何工作前，让我先谈谈使用了表查找的算法。访问一个表是一个非常快速的操作（除非此表与物理内存不匹配，这是另一种情况）。所以，如果某函数的值可以预计算并保存在一个表中，我们可以利用内存提高运行速度。isdigit 函数（宏）是这种交易的典型例子。原始的实现代码是：

```
inline bool IsDigitSlow(char c)
{
  return c >= '0' && c <= '9';
}
```

但是，如果我们注意到，对 isdigit 只能有 256 个不同的参数，我们就可以预先计算它们并保存在一个表中。让我们定义保存预计算值的类 CharTable。

```
class CharTable
{
public:
  CharTable();
  bool IsDigit(unsigned char c)const
  {
    return _tab [c];
  }
private:
  bool _tab [UCHAR_MAX + 1]; // 来自于<climits>头
};
CharTable::CharTable()
{
  for(int i = 0; i <= UCHAR_MAX; ++i)
  {
    // 使用慢速方法
    if(i >= '0' && i <= '9')
       _tab [i] = true;
    else
       _tab [i] = false;
  }
}
CharTable TheCharTable;
```

现在，我们可以迅速得知是否给定字符为一个数字，途径是调用：

```
TheCharTable.IsDigit(c)
```

实际上，isdigit 宏通常使用一个位字段静态初始化表的查找来实现（数字的位独立解释），其中，每个位与一个属性对应，例如为一个数字、一个空格、一个文字数字式字符等。

3.5.6　哈希表

哈希表数据结构是基于使用表查找来加速任意映射思想而实现的。为了实现我们的目的，我们感兴趣的是映射字符串到整数。我们不能直接使用字符串作为数组的索引。不过，我们可以定义一个辅助函数，将字符串转换成此索引。此函数称做哈希函数。于是，我们可以想像用两步来映射一个字符串到一个整数：对一个给定的字符串计算哈希函数的值，然后使用此结果作为访问数组的索引，而此数组包含有映射的预先计算值。

此类哈希称做完美哈希，它通常难以实现。在非理想情况下，我们通常满足于可能偶而将两个或多个不同字符串映射成同个索引的有缺陷哈希函数。这种情形称做冲突。因为冲突，哈希表将一个字符串映射成单个值而不是一个“短列表”，如图 3-15 所示。通过进一步搜索此列表，我们可以找到我们感兴趣的字符串以及它所映射的值。

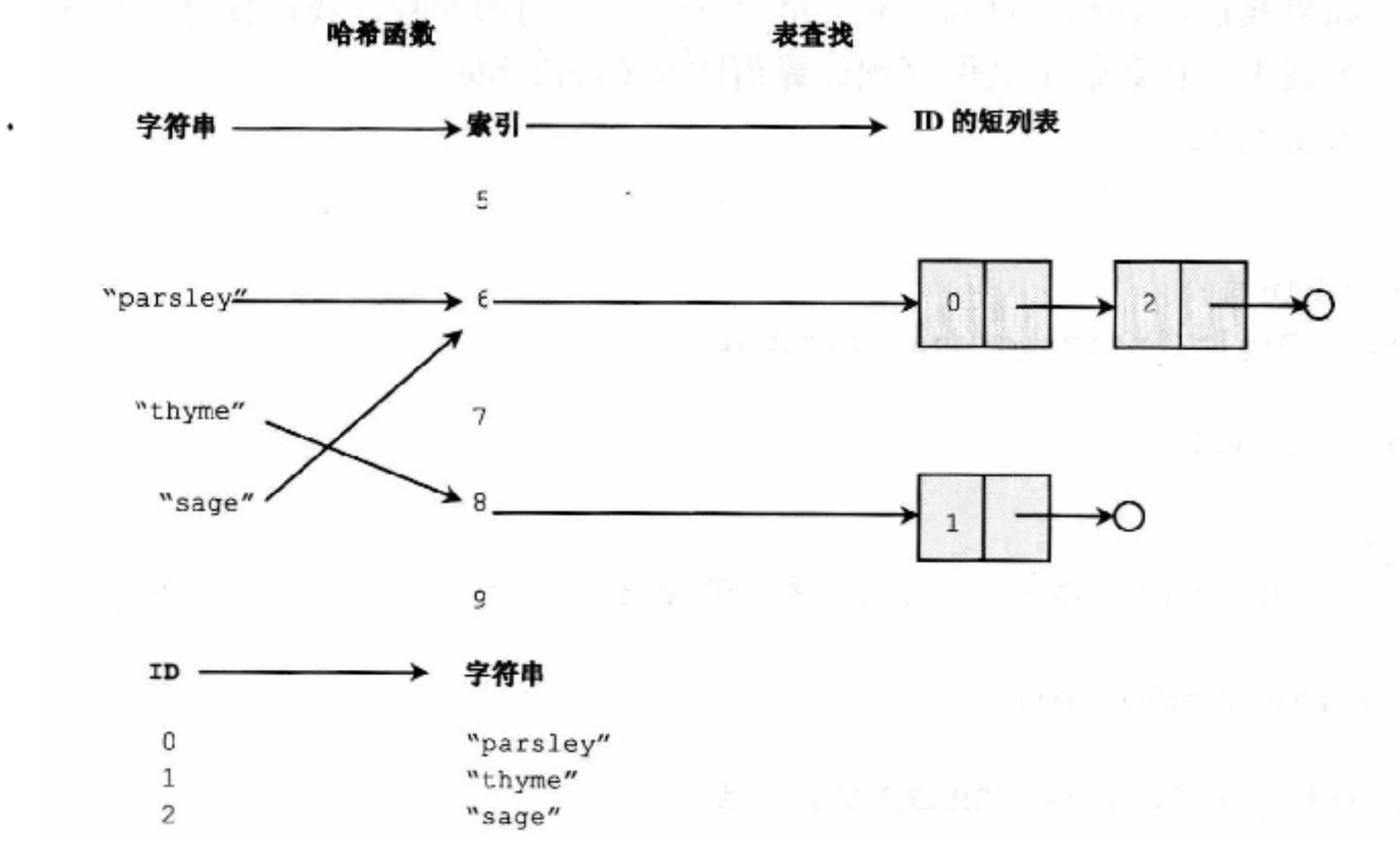


图 3-15　字符串“parsely”与“sage”的哈希函数的值相同。此冲突通过创建包含两个字符串 ID 的短列表来处理。这里有一个单独的字符串 ID 的映射

当被映射的字符串数量足够大时，此算法才有效。在 n 个字符串间直接的线性检索平均需要 n/2 次比较。另一方面，在哈希表中的检索平均需要一次比较（加上哈希函数的计算）。

例如，在字符串表中，我们最多可以保存 100 个字符串。在此表中直接找到给定字符串需要平均 50 次字符串比较。如果使用一个任意排列字符串的哈希函数，将这些字符串分布在 127 个项数组中，我们可望在第一次尝试中找到正确的字符串——只需要一次比较。这是一个重大的进步。

下面是类 HashTable 的定义。此表本身是一个列表数组（它们是我们所讨论的“短列表”）。其中的大部分包含 0 或 1 个元素。在很少发生冲突的情况下（即两个或多个字符串被哈希成

同一个索引），一些列表可能会更长。

```
const int sizeHTable = 127;
// 字符串哈希表
class Htable
{
public:
  // 返回可能的短列表
  List const & Find(char const * str)const;
  // 添加另一个 string->ID 映射
  void Add(char const * str, int id);
private:
  // 哈希函数
  int hash(char const * str)const;
  List _aList [sizeHTable]; // 一个(短)列表数组
};
//在可能包含有我们要找的字符串 ID 的哈希表中查找列表
List const & HTable::Find(char const * str)const
{
  int i = hash(str);
  return _aList [i];
}
void HTable::Add(char const * str, int id)
{
  int i = hash(str);
  _aList [i].Add(id);
}
```

选择哈希函数是重要的，因为我们不希望有太多的冲突。移位和相加算法是最好的短字符串随机生成器之一。

```
int HTable::hash(char const * str)const
{
  // 请不要有空字符串
  assert(str != 0 && str [0] != 0);
  unsigned h = str [0];
  for(int i = 1; str [i] != 0; ++i)
    h =(h << 4)+ str [i];
  return h % sizeHTable; // 余数
}
```

表达式 h << 4 等于 h 左移 4 位。在十六进制表示法中，左移 4 位等价于在数字后面补充一个 0。例如，0x3F 左移 4 位就是 0x3F0。移位的结果可能不适合于它所保存地方的整数类型（在此例中是无符号整数），在这种情况下，舍弃最左边的位。同时要注意，左移 4 位等价于乘以 16（通常情况下，移位等价于乘以或除以 2 的某次幂）。

哈希算法的最后一步计算 h 除以哈希表大小后的余数。此值可以直接用作 sizeHTable 数组项中的索引。表的大小也很重要。2 的幂产生的数字不可低估——它们能产生大量的冲突。

质数是最佳选择。通常 2 的幂加一或减去 1 就得到一个质数。在这里，$127 = 2^7 - 1$ 就是一个质数。

字符串“1”的哈希值为 114。它的计算过程参见表 3-4。

表 3-4　　字符串“One”的哈希值的计算

char	ASCII	h
'O'	0x4F	0x4F
'n'	0x6E	0x55E
'e'	0x65	0x5645

h 除以 127 的余数为 114，所以，字符串“1”的 ID 将保存在哈希表数组的偏移 114 处。

3.5.7　测试程序

下面是我们测试字符串表的方式：

```
int main()
{
  StringTable strTable;
  strTable.ForceAdd("One");
  strTable.ForceAdd("Two");
  strTable.ForceAdd("Three");
  int id = strTable.Find("One");
  std::cout << "One at " << id << std::endl;
  id = strTable.Find("Two");
  std::cout << "Two at " << id << std::endl;
  id = strTable.Find("Three");
  std::cout << "Three at " << id << std::endl;
  id = strTable.Find("Minus one");
  std::cout << "Minus one at " << id << std::endl;
  std::cout << "String 0 is "
          << strTable.GetString(0)<< std::endl;
  std::cout << "String 1 is "
          << strTable.GetString(1)<< std::endl;
    std::cout << "String 2 is "
          << strTable.GetString(2)<< std::endl;
}
```

calc4

3.5.8　练习

5．给动态堆栈添加一个私有 Shrink 方法。当_top 到达某低水位线时让 Pop 调用它。小

心：确保堆栈的大小不会在交换 Push 和 Pop 时导致分配与释放的交换。

6. 创建一个两级深度的字符串堆栈（char const *）。这一堆栈只记住最后两个字符串。当推入一个新字符串时，与最后一个字符串相邻的字符串被忘记，最后一个字符串取代它的位置，而且新字符串成为“最后一个”。Pop 将“与最后一个相邻”的字符串向“最后一个”字符串上移一个位置并在此位置保存 0。Pop 不返回任何值——使用 Top 来访问顶端字符串。

7. 根据 FIFO（first in, first out，先入先出）原则创建一个整数队列。方法 Put 和 Get 分别插入和读取元素。使用一个链表来实现。

8. 为链表创建一个序列生成器。创建另一个跟踪当前及以前项的序列生成器。在第二个序列生成器中实现方法 Unlink。此方法从被遍历的列表中删除当前项。测试特殊情况（删除第一个、删除最后一个、单元素列表等）。

9. 通过指针 _pNext 和 _pPrev 创建一个双链表项 DLink。实现 DLink 对象的 Unlink 方法。创建一个使用了循环双链表的 FIFO 队列，即最后的元素指向第一个元素、第一个元素反过来指向最后一个元素（如图 3-16 所示）。总是用封装自身的两个链接创建一个新 DLink。指向自身的指针可以通过关键字 this 来访问。例如，_pNext = this;将让 _pNext 指向当前 DLink。

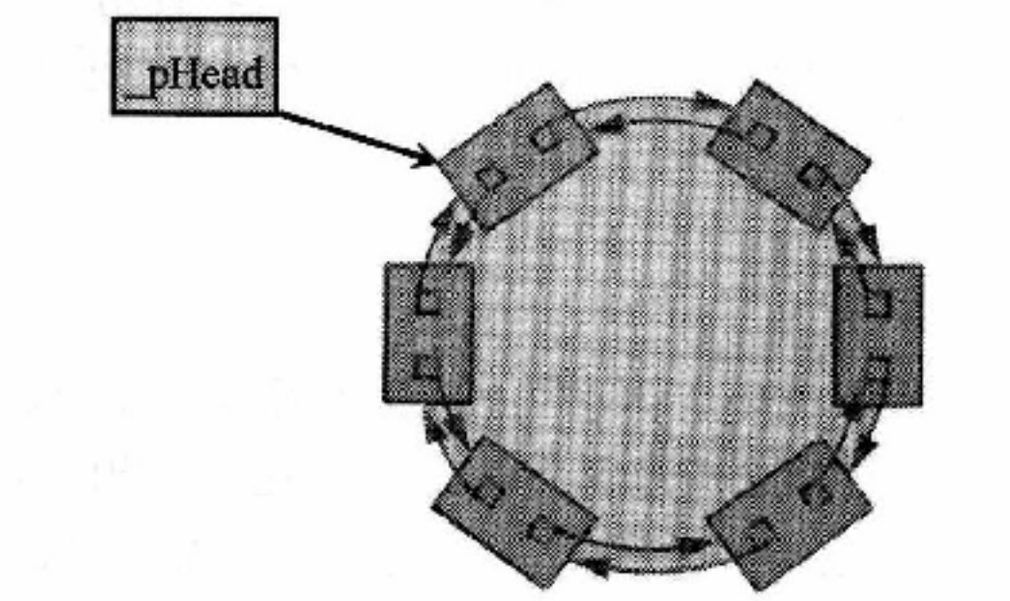


图 3-16 一个循环双向链表

10. 为哈希表设计和实现一个序列生成器。它必须遍历数组中的所有项并与所有的非空链表同步。

11. 使用上一个练习中的哈希表序列生成器设计和实现一个新的 StringTable，它使用了一个通过分配较大数组实现动态的哈希表，遍历旧哈希表并重新排列每一个元素形成新的数组。注意，新数组的哈希函数与旧哈希函数不同（除数使用不同的大小）。

第4章 多 态

多态意味着多种形态。收音机、磁带机及 CD 机形态各不相同，但它们具有相同的音频输出插口。你可以将耳机插入此插口来收听音乐，而与音乐是否来自于调制载波、录音带上的磁畴或可塑磁盘铝层上的一系列起凹点无关。从音乐爱好者角度看，所有这些设备都代表音频源。只要它们提供了相同的接口（音频插口），聆听音乐的过程就与之无关。

4.1 is-a 的意义

Star（星体）是一个 CelestialBody（天体），这意味着 Star 不仅具有 CelestialBody 的所有成员变量和方法，还具有自己特有的变量和方法。毕竟这只是相对直接嵌入的一种“语法糖块”。程序员可以嵌入一个 CelestialBody 作为 Star 的一个公有成员，除了啰嗦的 has-a 关系语义外，而且程序员可以模仿继承（事实上，这大概是编译器实现继承的方式）。

其间有一个重大区别：借助于继承，Star 可以作为一个 CelestialBody 传递。这意味着什么？它意味着一个需要指向 CelestialBody 的引用或指针的函数可以顺利接受一个指向 Star 的引用和指针（计算机语言学家称之为 Liskov 替换原理）。

下面举一个例子：类 BlackHole 有一个方法 Gobble，它接受任何 CelestialBody：

```
void BlackHole::Gobble(CelestialBody * pBody);
```

因为 Star 是一个 CelestialBody，所以，它可以被 BlackHole（黑洞）所 Gobbled（吞食）：

```
Star* pStar = new Star(1, 2);
TheBlackHole.Gobble(pStar); // 味道好极了
```

在 BlackHole 内如何对待 CelestialBody 呢？

```
void BlackHole::Gobble(CelestialBody* pBody)
{
  delete pBody;
}
```

它被销毁了!现在的一个大问题是：调用了谁的析构函数？一方面，我们知道我们销毁了一个 Star，所以可以看到打印输出“Destroying a star…”。而另一方面 BlackHole 不知道它正吞食一个 Star。它希望看到“Destroying celestial body…”。而且在编译时，编译器不知道给 BlackHole 发送什么。请想象这样一种情形：

```
Star * pStar = new Star(1, 2);
Planet * pPlanet = new Planet(3, 4);
TheBlackHole.Gobble(pStar); //味道好极了
TheBlackHole.Gobble(pPlanet); //味道好极了
```

在这两种情况下，调用了同一个方法 Gobble，而且执行的是同样的代码，所以，很显然编译器将在 Gobble 内只插入对 CelestialBody 的析构函数的调用（或扩展内联）。编译器基于指针的类型分派调用。请注意，用 Star 或 Planet 覆盖 CelestialBody 的方法同样如此。如果 Gobble 调用以上任何一个方法，编译器将调用 CelestialBody 的实现代码，而不是调用 Star 或 Planet 提供的代码。

此方案开销少，而且足以应付许多情况。不过，相对于少量的额外开销，C++提供了一种非常强大的功能，这就是多态（polymorphism）。如果你想知道 BlackHole 内的不同情况，根据落于其中的对象的不同，你必须声明 CelestialBody 的析构函数为 virtual。当一个虚函数在一个派生类中被覆盖时，对此函数调用的分配由对象所指向的实际类型完成，而不是由指针类型完成。

4.1.1 谈谈实现

显然，为了让多态发挥作用，对象本身必须保存一些有关实际类型的信息。基于这一类型，将这一调用分配给方法的适当实现过程。事实上，对象可以保存一个指向分配表的指针，虚方法通过它进行调用。

它的工作方式是：至少有一个虚方法的类（一个多态类）有一个隐藏成员，它是虚表的指针。虚表（vtable）包含相对此类虚成员函数的地址（如图 4-1 所示）。当通过一个此对象的指针或引用来进行调用时，编译器生成解除此对象虚表的指针，并使用保存在此表中的成员函数地址进行间接调用（这些内容有些超前，与函数和方法的指针一样，它们都将在本书后面章节讲解）。

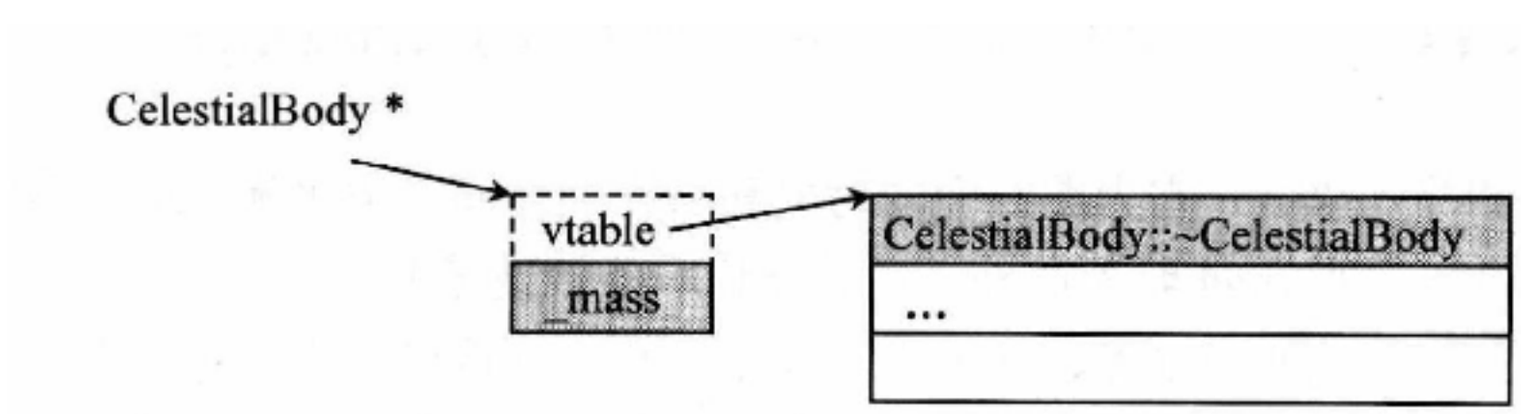


图 4-1 天体的不可见成员——vtable 的指针。vtable 中的第一项为一个指向函数的指针，这一函数为天体的析构函数

覆盖了虚方法实现过程的派生类的对象有自己指向不同虚表的指针（如图 4-2 所示）。在此表中，对应于特定方法的项包含覆盖成员函数的地址。

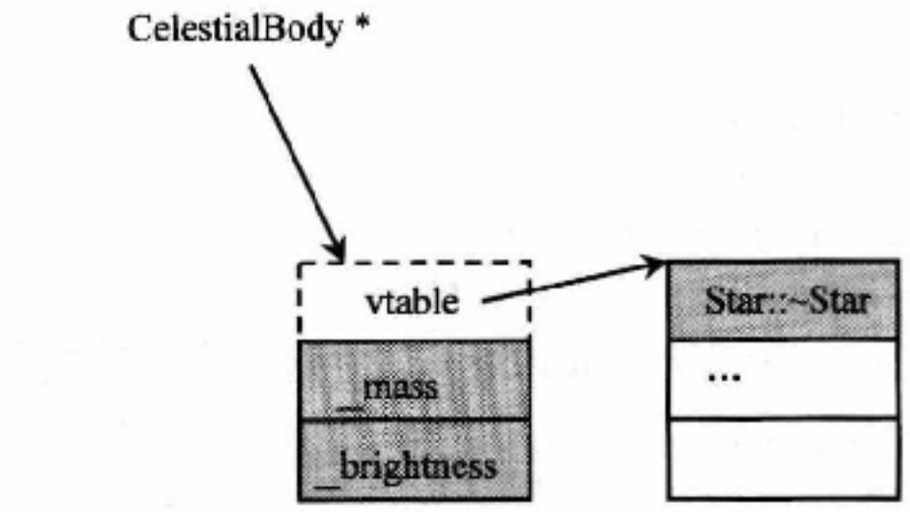


图 4-2　指向不同 vtable 的星体的 vtable 指针。它的第一个项也是一个指向用作析构函数的指针，只是这一次它是一个星体的析构函数

4.1.2　开销

在你决定在程序中使用虚函数之前，你必须考虑大小和速度因素。多态的空间开销为一个对象（类的实例）一个指针。还有另外的 vtable 各个类开销，但这关系不大。当处理少量的相对较大的对象时，一个对象一个指针的开销不是很多。但是，当处理大量轻量级对象时，其开销会猛增。

调用一个虚函数比直接调用速度慢，而且比使用内联函数速度要慢得多。再次说明，当你调用一个重量级的成员函数时，可以安全地忽略此开销，但是在虚函数中利用 AtEnd 之类的内联函数可能大大减慢循环速度。

所以，不要利用手边的虚析构函数创建类对象（它是所有对象之源）。也可能通过某些种类的类 ID 的多个整数域添加一些额外的编译调试设备等——费时的输入!不要尝试让它作为所有类的源。事实上，如果你听到有人抱怨 C++有多慢，他（或她）大概误用了 C++从 Smalltalk 引入的这些特征。

倒不是 Smalltalk 语言不完善。当大小和速度影响不大时，Smalltalk 在许多方面优于 C++。它是一种真正继承了 C 精髓的面向对象语言，统一了内置类型与用户自定义类型，这一点比 C++做得好。它有一个单根源对象层次。所有的方法为虚方法（你甚至可以在运行时覆盖它们）。

Java 是 C++的简化语言，它力图在面向对象和操作性之间寻求平衡。Java 中的所有方法都是虚方法，当显式声明 final 时才例外（而且它们不能被覆盖）。

如果速度和大小是我们必须考虑的因素，则使用 C++并恰当运用多态。设计完美的多态类所生成的 C++代码至少与等价的 C 代码同样快（从维护的角度讲，则更胜一筹）。

C 语言程序员的一般经验是检查 switch 语句和复杂的条件句。而在 C++中，可以很自然地在其位置使用多态。

4.2 解析树

虚成员函数、虚析构函数、纯虚函数、受保护数据成员。

tree1

我将用一个非常有用的数据结构例子（算术树）演示多态的使用。算术表达式可以转换成一个树结构，它的节点为算术运算符、它的树叶节点为数字。图 4-3 显示了对应于表达式 2 *（3 + 4）+ 5 的算术树。从根到叶对它进行分析，我们首先碰到“加”节点，它的子代为两个相加的项。左边子代为两个因子的积。左边因子为数字 2，右边因子为 3 和 4 的和。上一级加节点的右边子代为 5。注意，树表示法中不需要任何小括号或运算符优先级的知识。它所描述的计算过程是唯一的。

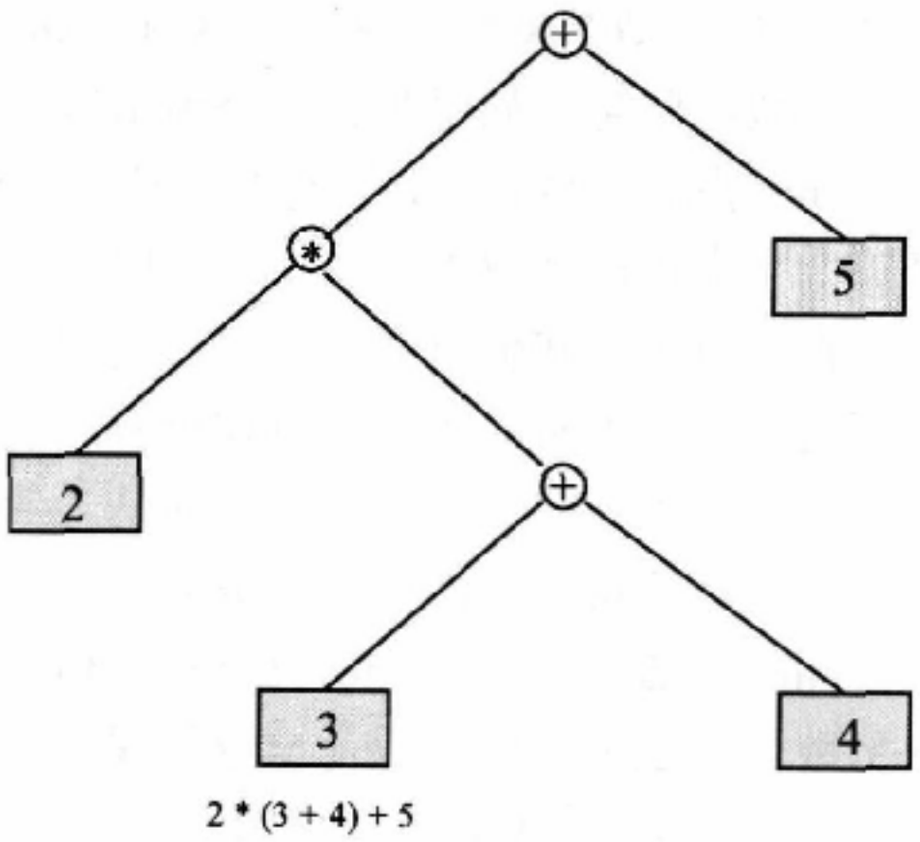


图 4-3 对应表达式 2 *（3 + 4）+ 5 的算术树

我们将算术树的所有节点表示为从类 Node 继承的对象。Node 的直系后代为 NumNode，它表示一个数，同时 BinNode 表示一个二元运算符。为简单起见，我们将限制从 BinNode 派生两个类：AddNode 和 MultNode。图 4-4 为我们已经描述过的类层次 I。抽象类为不能实例化的类；它们只能作为其他类的父类，稍后我将更详细地解释它。

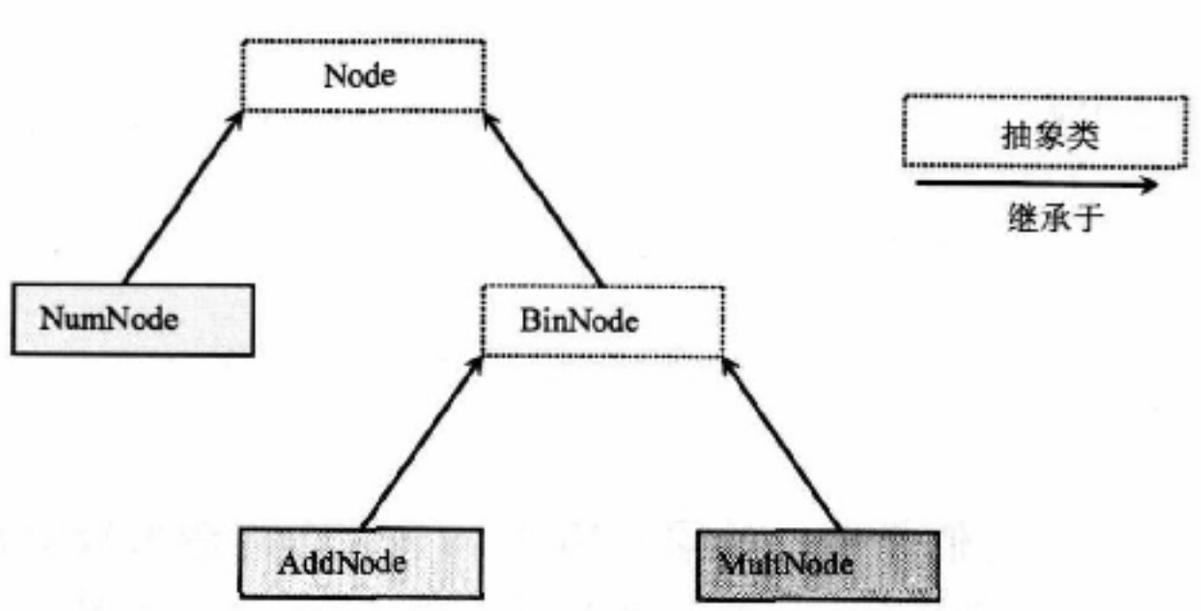


图 4-4 节点的类层次

我们将对一个节点执行什么样的操作？我们可以计算其值，某些时候还可以销毁它。Calc 方法返回一个作为计算节点值的双精度数。当然，一些节点的计算可能包含对其子节点的一些递归计算。此方法是 const，因为它不会改变本身节点。每个节点类型必须提供它自己的 Calc 方法，

所以，我们声明此方法为 virtual。但是对于任意节点，没有“缺省”的 Calc 方法实现方式。一个没有具体实现（继承而来或其他）的函数称做纯虚函数。这就是 Calc 声明中=0 的意义。

```
class Node
{
public:
  virtual ~Node(){}
  virtual double Calc()const = 0;
};
```

一个包含一个或多个纯虚函数的类称做抽象类，它不能实例化（此类没有对象可创建）。只有从一个抽象类派生的类以及为所有纯虚函数提供了实现代码的类才能实例化。请注意，上例中的算术树有 AddNodes、MultNodes 和 NumNodes 的实例，但没有 Nodes 或 BinNodes 的实例。

一般经验是，如果类有一个虚函数，它可能同时需要一个虚析构函数——一旦我们决定为 vtable 指针支付开销，所有的后续虚函数就不会增加此对象的大小。所以，在这种情况下，添加一个虚析构函数不会增加任何巨大的开销。

在此例中，我们可以期望一些后代节点必须在其析构函数中销毁它们的子节点，所以，我们需要一个虚析构函数。析构函数可以是一个纯虚函数，但是我们不得不在每个派生类中覆盖它，而不管此类需要与否。这是为什么我让基类构造函数体为空的原因（即使我使它内联，编译器也将为它创建一个函数体，因为编译器需要为它在虚表中固定一个指针）。

NumNode 保存一个 double 值，此值在它的构造函数中被初始化。NumNode 还覆盖了 Calc 虚函数。在这里，Calc 只简单返回保存在此节点中的值。

```
class NumNode: public Node
{
public:
  NumNode(double num): _num(num){}
  double Calc()const;
private:
  const double _num;
};
double NumNode::Calc()const
{
  std::cout << "Numeric node " << _num << std::endl;
  return _num;
}
```

BinNode 有两个子代，它们指向（抽象）Node。它们在构造函数中初始化并在析构函数中删除——这是我可以让它们成为常量指针的原因。Calc 方法仍然是一个从 Node 继承的纯虚函数；只有 BinNode 的后代才知道如何实现它。

```
class BinNode: public Node
{
public:
  BinNode(Node * pLeft, Node * pRight)
    : _pLeft(pLeft), _pRight(pRight){}
```

```
  ~BinNode();
protected:
  Node * const _pLeft;
  Node * const _pRight;

};
BinNode::~BinNode()
{
  delete _pLeft;
  delete _pRight;
}
```

在这里，你第一次见到了多态的优点。一个二元节点可以有任意节点的后代。每个后代可以为数字节点、加法节点或一个乘法节点。后代有 9 种可能的组合——为每种组合配备一个单独的类是愚蠢的（请想象一下 AddNodeWithLeftMultNodeAndRightNumberNode）。我们别无选择，只有接受并作为 Node 的更一般指针保存指向后代的指针。然而，当我们通过它们调用析构函数时，我们需要调用不同的函数来销毁不同的 Node。例如，AddNode 的析构函数与 NumNode 不同（后者没有）。这是我们为什么必须使 Node 的析构函数为虚函数的原因。

注意，BinNode 的两个数据成员不是私有（private）——它们为受保护（protected），此级别比私有稍弱。一个私有数据成员或方法不能从给定类（或友元）外部的任何代码中进行访问——甚至从派生类的代码也不能访问。如果我们让_pLeft 和_pRight 成为私有，我们将不得不通过公有方法来设置和读取它们。这就等于将它暴露给所有人了。通过让它们受保护，我们可以让从 BinNode 派生的类操纵它们，但同时，要阻止其他任何人这样做。表 4-1 列出了所有 3 个访问修饰符的简短描述。顺便说一下，在声明继承时会用到同样的访问修饰符。到目前为止，最有用的继承类型是 public，而且它也是我们一直所使用的，但如果你为了限制对基类的访问，你可以选择使用其他两种继承类型。

类 AddNode 派生于 BinNode。

```
class AddNode: public BinNode
{
public:
  AddNode(Node * pLeft, Node * pRight)
    : BinNode(pLeft, pRight){}
  double Calc()const;
};
```

表 4-1　　3 个访问定义符的意义

访问定义符	谁可以访问这一成员
public	任何人
protected	类本身、其友元及派生类
private	只有类本身及其友元

AddNode 提供了自己的 Calc 实现代码。在这里，你又一次见识了多态的优点。我们让子节点计算它们自己。因为 Calc 方法为 virtual，所以它们将根据实际对应的类完成适当的任务，而不会根据指针（Node *）的类。调用 Calc 的两个结果是相加并将和值返回。

```
double AddNode::Calc()const
{
  std::cout << "Adding\n";
  return _pLeft->Calc()+ _pRight->Calc();
}
```

请注意 AddNode 的方法是如何直接访问它的父受保护数据成员_pLeft 和_pRight 的。再次说明，如果将它们声明为 private，编译器将认为此访问是一个错误。

为了完整性，下面提供了 MultNode 的实现代码及一个简单的测试程序。

```
class MultNode: public BinNode
{
public:
  MultNode(Node * pLeft, Node * pRight)
    : BinNode(pLeft, pRight){}
  double Calc()const;
};
double MultNode::Calc()const
{
  std::cout << "Multiplying\n";
  return _pLeft->Calc()* _pRight->Calc();
}
int main()
{
  //(20.0 +(-10.0))* 0.1
  Node * pNode1 = new NumNode(20.0);
  Node * pNode2 = new NumNode(-10.0);
  Node * pNode3 = new AddNode(pNode1, pNode2);
  Node * pNode4 = new NumNode(0.1);
  Node * pNode5 = new MultNode(pNode3, pNode4);
  std::cout << "Calculating the tree\n";
  // 通知根计算其自身
  double x = pNode5->Calc();
  std::cout << "Result: " << x << std::endl;
  delete pNode5; // 及所有子代
}
```

你认为你能不用多态来编写出更有效的代码吗？请仔细想一想。如果你仍然没能信服，去应用 C 来亲身实践一下。

4.2.1 谈谈 C

tree2

C 中没有虚函数，那它是如何解决同样问题的呢？一个好的途径是尝试人工实现一个虚

表，然后通过函数指针来调用。你对这样工作的频繁度将感到惊讶。在 C 中这叫做面向对象的编程。更传统的途径是定义一个 struct Node，它有足够的域来容纳二元运算符以及数字节点的需要。节点身份将保存在单独的域中。

```
#define NUM_NODE 1
#define ADD_NODE 2
#define MULT_NODE 3
struct Node
{
  /* Node 类型 */
  int type;
  /* 仅在数字节点中使用 */
  double val;
  /* 仅在二元节点中使用 */
  struct Node * pLeft;
  struct Node * pRight;
};
```

C++中的虚函数将被转换成多重条件句，如下面的 switch 语句（对 switch 的描述参见 5.1.3 节的内容）。

```
double Calc(struct Node * pNode)
{
  double x;
x = 0.0;
switch(pNode->type)
{
case NUM_NODE:
  x = pNode->val;
  break;
case ADD_NODE:
  x = Calc(pNode->pLeft)+ Calc(pNode->pRight);
  break;
case MULT_NODE:
  x = Calc(pNode->pLeft)* Calc(pNode->pRight);
  break;
}
return x;
}
```

tree3

显然，当此方案应用于包含上百个节点的树时开销太大。此时，C 程序员就要使出他的或她的有力秘密武器了。在这种情况下，类型安全是首先要尝试的——使用强制类型转换（让人觉得好笑）。程序员不是将所有的一切装进一个单独结构（struct），而是创建许多结构：

```
struct Node
{
```

```
  int type;
};
struct NumNode
{
  int type;
  double val;
};
struct BinNode
{
  int type;
  struct Node * pLeft;
  struct Node * pRight;
};
```

甚至有些程序员使用一个字节来保存类型——以速度交换空间（对于许多处理器来说，取回一个以单词排列的值比取回一个以字节排列的值要快）。函数 Calc 带有一个指向 Node 的指针，基于它的类型域将它转换成适当的指针（事实上，程序员可以去掉最后的类型安全伪装并传递 Calc 一个空指针）。

```
double Calc(struct Node * pNode)
{
  double x = 0.0;
  switch(pNode->type)
  {
    case NUM_NODE:
      x =((struct NumNode *)pNode)->val;
      break;
    case ADD_NODE:
    {
      struct BinNode * pN =(struct BinNode *)pNode;
      x = Calc(pN->pLeft)+ Calc(pN->pRight);
      break;
    }
    case MULT_NODE:
    {
      struct BinNode * pN =(struct BinNode *)pNode;
      x = Calc(pN->pLeft)* Calc(pN->pRight);
      break;
    }
    default:
      printf("Bad node type\n");

  }
  return x;

}
```

我还没有解释类型转换，此时解释它非常勉强，时机尚不成熟。强制类型转换意味着欺骗编译器。一个对象声明为指向 Node 的指针，而你告诉编译器将它当成一个指向 BinNode 的指针。通过这种方式，你骗过了编译器类型检查的精确机制。现在，轮到程序员来确保此指针实际是指向 BinNode 而不是指向其他对象。

注意，甚至“构造函数”CreateNumNode 也必须使用类型转换。析构函数将既使用类型转换又使用条件句。

```
struct Node * CreateNumNode(double value)
{
  struct NumNode * pNode = malloc(sizeof(struct NumNode));
  pNode->type = NUM_NODE;
  pNode->val = value;
  return(struct Node *)pNode;
}
```

以上 C 解决方案与 C++多态方案相比如何？至于数据结构的大小因素，C 没有提供相应的节省办法。事实上，第一个 C 解决方案实质上更消耗内存。

至于执行的速度因素，我们就要比较多分支条件句加上一个直接调用与通过 vtable 的双重间接调用。比较的结果取决于编译器优化条件句的程度。在此例中，因为 case 标签句是连续的，所以编译可能会创建一个由类型值索引的跳转表。在执行间接跳转前，编译器必须检查是否索引在界内（如果越界，编译器将跳到缺省标签）。所以，事实上，我们拥有一个条件句、一个间接跳转及一个函数调用。在 C++中，我们应用了一个双重间接函数调用。C++编译器通常通过将传递的指针放在一个寄存器中进行优化，所以，调用序列可能在 C++中更简单。底下一行没有对两个版本进行实际的计时，因此没法知道哪一个版本更快。

至于正确性及维护的难易程度因素，你可以想像一下，给每种实现方式添加一种新节点类型后的样子。同时注意，即使我们用 C++编写的程序越来越复杂，但我们还没有感到需要类型转换或开关语句（switch）的帮助。

即使是 C 程序员也会承认使用类型转换是最后一招儿。C++程序员在使用开关语句时的感觉也是如此。

仅在不能使用多态时使用开关语句。

4.2.2 练习

1．设计一个程序，通过删除所有能被 2、3、5 和 7 整除的数来计算 100 以内的所有质数。创建一个抽象类 Sieve，它包含一个纯虚方法 NextNumber。实现 SourceSieve，此方法按顺序重复遍历从 1 到 100 的所有数字。实现 Sieve2，在它的构造函数中带有一个对 Sieve 的引用，而且当请求它给出下一个数字时，它将一直从它的源 Sieve 读取数字，直到发现一个不能被 2 整除的数并返回此数（但是它必须返回 2，因为 2 为质数）。重复用 3、5 和 7 来除。

在 main 中作为本地对象创建所有的 Sieve，通过传递其中的一个给另一个的构造函数将它们链接起来，并打印顶端 Sieve 返回的所有的数字。什么样的 Sieve 顺序最佳？

2. 创建一个抽象类 Command，它有两个纯虚方法 Execute 和 Undo。对应于所有的可能计算器输入，为基于堆栈的计算器创建命令类。对每个用户输入构建一个相应的 Command 对象，执行它（由 Execute 方法）并将它推入一个两级深度的 Command 堆栈中。为计算器添加撤销（u-命令）。u-命令从命令堆栈中弹出最后的 Command，并调用其虚方法 Undo。

第5章 一个小型软件项目

当你编写一段程序时，你不会问自己“我如何能使用一种特殊的语言功能？”，而会问“哪一种语言功能可以帮助我解决难题？”

5.1 开始一个软件项目

自顶向下设计和实现。扫描器、自顶向下递归解析器、符号表。从函数返回对象的分配和拥有。函数指针、开关语句（switch）。

程序员很少对自己编写的代码完全满意。所以，当我编写完此章的程序后（一个简单的解析-计算器），我立即觉得我可以做得更好。我感到程序的顶层视图不是很清晰。解析树的节点需要访问 Store（计算器的“记忆”）、符号表（打印有意义的错误信息），甚至函数表（内置函数求值）这样的组件。我让这些对象全局化，因而所有人都知道它，或给它们或所有的树节点传递引用。这些对象的构建顺序是重要的，所以，我们觉得它们应该组合成一个高级别的对象 Calculator。重新设计和重新编写整个程序的愿望很强烈。也许在经过了 n 次迭代后，我才能得到接近理想的代码？

最后我决定不这样做。编写程序的一部分工作就是先写出不太理想的方案，然后再进行完善。如果我认为提供了最好的方案，然后再反过来改成适应自顶向下的设计和实现过程，这就是在说谎。所以，这就是创建一个程序的实实在在、真实的过程。

5.1.1 设计规范

计算器程序的目的是接受用户提供的算术表达式、求其值并显示结果。表达式由一个简单的自顶向下递归下降解析器[1]（如果你对此方面知识不了解，不要害怕——这只是后面将详

[1] 从原理上讲，解析器的类型是一个实现细节，外部规范通常不会提及这样的细节。不过在这里，解析器的类型影响着语法的选择，这是外部规范的一部分。

细描述的一个名称而已）分析。解析器的目标是将输入字符串转换成一个算术树。解析器所接受的语法定义如下：

解析器寻找一个表达式。

1. 表达式
 a. 是一个后面有加号或减号的*项*，加号或减号的后面又有另一个*表达式*。
 b. 如果表达式不含任何加号或减号，它等于此*项*。
 c. ***表达式***也可以为一个*项*加一个等于号，再加一个*表达式*。
2. 一个**项**是
 a. 被另一个*项*乘或除的*因子*。
 b. 如果一个项不包含任何乘或除运算符，它等于此*因子*。
3. 一个**因子**可以是
 a. 一个数字
 b. 一个对应于某变量的标识符
 c. 后面带有一个因子的减号（一元减）
 d. 小括号中的整个**表达式**

例如，表达式 1 + x *（2 - y）解析过程如图 5-1 所示。

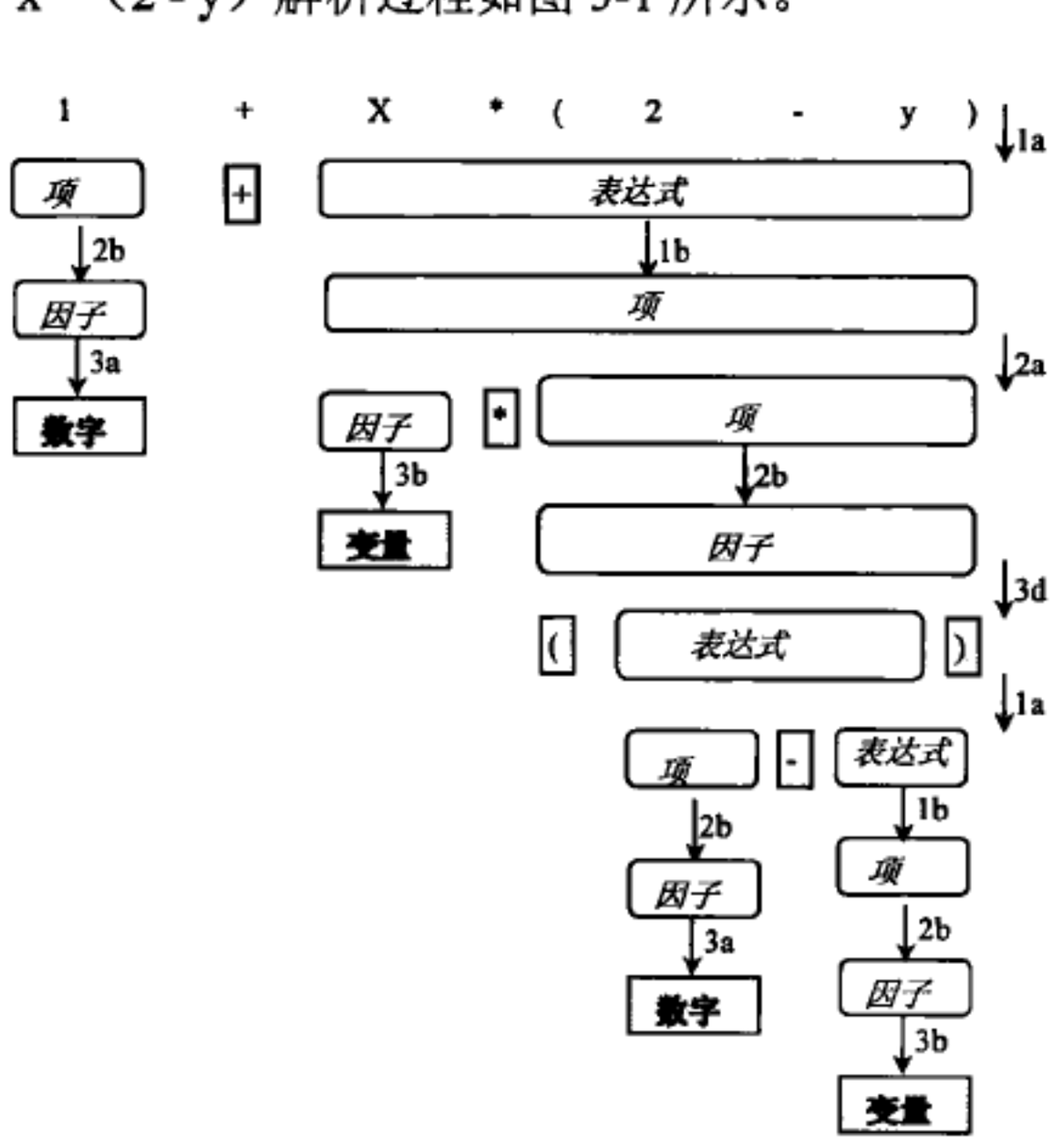


图 5-1　算术表达式的解析。带有与语法规则对应数字的箭头用于每一步的解析

语法是简单和自然的，但它有一个缺陷——算术运算符为右结合而不是左结合。这就是说，a - b + c 被解析成 a –（b + c），这并不是我们所期望的。应用标准途径可以修

正此语法或修改此解析器，但这超出了本节的讨论范围（我们将在后面章节重新讨论此主题）。

因为我们想让计算器能使用变量，所以我们需要一个符号表。用符号表记住变量的名字。我们将重用第 3 章中所描述的字符串表。

我们还要能给变量赋值。通过扩展表达式的定义，让它包含以上的条款 1.c 来完成。

并不是每个项均适合放在赋值句的左边。它必须为左值（lvalue）——即可以放在赋值号左边的值。在这里，唯一允许的左值类型是对应于变量的标识符。同样，放在赋值号右边的对象称做右值（rvalues）。在这里，所有的表达式都是有效的右值[1]。

我们还要引入一个扫描器，它将输入字符串转换成一系列记号。它还将识别算术运算符、数字及标识符。按这种方式，解析器更容易实现记号序列与语法定义间的匹配。

5.1.2 简短实现代码

在这里，我将遵循自顶向下的实现策略。这意味着，我将首先运行此程序，然后再实现它（实际上，我一创建了所有顶层类的代码片断就立即运行它）。

本章的源代码以 Code Co-op 脚本的形式提供。Code Co-op 的使用说明请参阅本书“前言”。项目名字为 Calculator（见边注）。有时，我会简短地谈及一些相关技术。你可以在光盘中找到这些相关技术的源代码。边注指出了对应的目录名（不是脚本号）。

Project Calculator

让我们从扫描器开始。它由文本（准确地说是一行文本）缓冲器构建而成。扫描器用一个指针指向此缓冲器，然后可以从左到右扫描此缓冲器并将它们转换成记号。暂时让 Scanner 的构造函数代码只声明它的存在并打印缓冲器的内容。

```
class Scanner
{
public:
  Scanner(char const * buf);
private:
  char const * const _buf;
};

Scanner::Scanner(char const * buf)
  : _buf(buf)
{
  std::cout << "Scanner with \"" << buf << "\""
     << std::endl;
}
```

[1] lvalue 也可能从“location value”而来，因为它代表了一个值可以存放的位置；而 rvalue 来自“read value”，因为从它可以读取一个值。

注意，为了在字符串字面值中包含一对双引号，我们必须在前面加一个转义字符——反斜杠。反斜杠本身不会变成字符串的一部分，它在这里仅仅是告诉编译器后面的引号不是字符串的结尾。如果你确实需要在字符串中加一个字面反斜杠，你必须再写一遍此反斜杠。

SymbolTable 代码片断此时的用途不大。假定它有一个构造函数。

```
class SymbolTable
{
public:
  SymbolTable(){}
};
```

解析器需要访问扫描器及符号表。它将解析从扫描器读取的记号并评估得到的树。这由方法 Eval 来完成并且返回与解析结果相关的状态码。我们用一个枚举设置了 3 种可能的状态：stOk、stQuit 和 stError。enum 是一种只可以包含若干预定义值的整数类型。这些值被赋予符号名字并由程序员或编译器初始化为具体值。在这里，我们并不真正关心不同状态所对应的值，所以，将初始化工作留给编译器完成。对 Eval 的返回类型应用枚举而不使用整数的优点是为了实现更严格的类型检查。它还阻止我们返回枚举三个定义值以外的其他任何值。

```
enum Status
{
  stOk,
  stQuit,
  stError
};

class Parser
{
public:
  Parser(Scanner & scanner, SymbolTable & symTab);
  ~Parser();
  Status Eval();
private:
  Scanner      & _scanner;
  SymbolTable  & _symTab;
};

Parser::Parser(Scanner & scanner, SymbolTable & symTab)
  : _scanner(scanner), _symTab(symTab)
{
  std::cout << "Parser created\n";
}
```

```
Parser::~Parser()
{
  std::cout << "Destroying parser\n";
}

Status Parser::Eval()
{
  std::cout << "Parser eval\n";
  return stQuit;
}
```

下面是 main 过程。你可以从中看到程序的顶层设计在发挥作用。SymbolTable 的生存期必须与程序的生存期相同，因为它必须记住在一次对话中由用户引入的所有变量名。扫描器和解析器可以在每次输入一行文本时得以创建。解析器没有声明必须保存从一行文本到另一行文本。如果解析器碰到一个新的变量名，它将在具有更多长生存期的符号表中保存此名字。

```
const int maxBuf = 100;

int main()
{
  char buf [maxBuf];
  Status status;
  SymbolTable symTab;
  do
  {
    std::cout << "> "; // 提示
    std::cin.getline(buf, maxBuf);
    Scanner scanner(buf);
    Parser parser(scanner, symTab);
    status = parser.Eval();
  } while(status != stQuit);
}
```

在程序的主循环中，通过 cin 的 getline 方法从标准输入获取文本。与使用>>提取单个单词或数字相比，这是一种提取整行文本的标准方式。扫描器构建于文本行，解析器创建于扫描器。于是调用解析器的 Eval 方法来解析和求表达式的值。只要 Eval 返回的状态不同于 stQuit，整个过程就会重复执行。这里所使用的 do/while 循环与 while 循环不一样，它的循环体总是至少执行一次。

编译和运行此程序，可以证实以上概念的正确性。

5.1.3 扩展程序片断



Script 1

将 Scanner 的第一个程序片断扩展成完整的代码。扫描器将输入字符串转换成一系列记

号。它的工作方式像序列生成器一样。只要 Parser 需要一个记号，它就会向 Scanner 请求当前记号。当解析此记号时[1]，Parser 调用 Scanner::Accept 来接受它。Accept 方法进一步扫描字符串，尝试识别下一个记号。

我们可以方便地将一套定义明确的记号放到枚举 EToken 中（我们暂时只使用有限数量的记号，但根据需要我们将逐渐增加）。

```
enum Etoken
{
  tEnd,
  tError,
  tNumber,
  tPlus,
  tMult
};
```

我还要将 Scanner 识别为数字的部分输入字符串转换成浮点值。这由 Number 方法完成，它只在当前记号为 tNumber 时才调用（参见那里的断言）。注意类型 char const * const 的使用，它是一个指向常量字符串的常量指针。此指针在构造函数中初始化，以后再也不会更改。它的内容也是只读的。

```
class Scanner
{
public:
  Scanner(char const * buf);
  EToken Token()const { return _token; }
  void Accept();
  double Number()
  {
    assert(_token == tNumber);
    return _number;
  }
private:
  void EatWhite();
  char const * const _buf;
  int                _iLook;
  EToken             _token;
  double             _number;
};
```

Scanner 的构造函数除了初始化所有的成员变量外，还调用方法 Accept。Accept 识别第一个变量记号并将索引_iLook 放置到缓冲器的识别部分之后。

```
Scanner::Scanner(char const * buf)
  : _buf(buf), _iLook(0)
{
  std::cout << "Scanner with \"" << buf << "\""
```

[1] 即使解析器识别了某个错误，它通常仍然接受这一记号，因此解析不会进入无限循环。

```
        << std::endl;
  Accept();
}
```

EatWhite 为辅助函数，它略过输入中的空格。

```
void Scanner::EatWhite()
{
  while(std::isspace(_buf [_iLook]))
    ++_iLook;
}
```

Accept 只是一个庞大的 switch 语句。根据当前缓冲器中的字符_buf[_iLook]，流程控制传递到 switch 语句中的适当的 case 标签。我们暂时只实现加法和乘法运算符的识别。当识别以上运算符之一时，将_token 变量初始化为对应的枚举值，并将变量_iLook 增加 1。

阿拉伯数字的识别由 case（事例）语句的穿越特性来实现。除非 switch 语句中有一个显式的 break，否则，控制将传递到下一个 case。所有的数字事例都将穿越，到达 case '9'以后的代码。将以数字开始的字符串转换成一个整数（以后，我们修改此部分程序使其识别浮点数），并将_iLook 放在最后的数字之后。

当没有事例与传递给 switch 的值匹配时执行 default 事例。

```
void Scanner::Accept()
{
  EatWhite();
  switch(_buf[_iLook])
  {
  case '+':
    _token = tPlus;
    ++_iLook;
    break;
  case '*':
    _token = tMult;
    ++_iLook;
    break;
  case '0': case '1': case '2': case '3': case '4':
  case '5': case '6': case '7': case '8': case '9':
    _token = tNumber;
    _number = std::atoi(&_buf [_iLook]);
    while(std::isdigit(_buf [_iLook]))
      ++_iLook;
    break;
  case '\0': // 输入结尾
    _token = tEnd;
    break;
  default:
    _token = tError;
    break;
  }
```

```
}
```

你可能会问，“此 switch 语句可以使用恰当的多态来代替吗？”，我们实在不知道程序员如何才能做到这一点。困难在于输入的数据流（来自缓冲器的字符）是不确定的。当我们努力确定输入的意义时（确定它的形式），最佳途径就是应用一个多分支条件结构，在这里就是 switch 语句[1]。实际上，在 C++中使用 switch 语句的唯一完全合法场合是处理不确定输入。解析用户输入或保存在文件中的数据，或者处理外部事件都需要 switch 语句或多分支条件句。

哑元解析器暂时用一个 for 循环来实现，它逐个读取记号并打印适当的信息。

```
Status Parser::Parse()
{
  for(EToken token = _scanner.Token();
      token != tEnd;
      _scanner.Accept())
  {
    token = _scanner.Token()
    switch(token)
    {
    case tMult:
      std::cout << "Times\n";
      break;
    case tPlus:
      std::cout << "Plus\n";
      break;
  case tNumber:
    std::cout << "Number: " << _scanner.Number()
              << std::endl;
    break;
  case tEnd:
      std::cout << "End\n";
      return stQuit;
  case tError:
    std::cout << "Error\n";
    return stQuit;
  default:
    std::cout << "Error: bad token\n";
      return stQuit;
    }
  }
  return stOk;
}
```

再一次编译和测试此不完整程序。我们可以看到流程控制及少数记号的正确识别。

[1] 在某些情况下，我们可以使用表驱动途径越过条件。例如，因为只有 256 个可能的字符，我们可以定义一个有 256 个项的表，再用当前输入字符来索引它。其中的项可以是为处理给定字符类型特定设计的函数指针（我们将在后面谈及）。

最终程序？绝对不是的!

程序完善永无止境。让我再次重复：

程序完善永无止境。

那么，我们如何知道什么时候终止呢？事实是：我们并不知道!有时，我们厌倦了此程序；有时，我们开始一个新的有趣的项目并放弃现在的程序（当然，我们不会放弃所讨论的程序）。将交付日期提前（即使我们稍有超时）、一个经理告诉我们停止添加新的功能并发布此产品，这一切都可能导致完善工作的暂停。

明白这一点后，让我们再次回到“伪”最终版的计算器上，进一步演示如何扩展程序片断创建工作正常的程序。

5.2 扫描器

指针的指针类型、通过引用传递简单类型、C++标识符、传递缓冲器。

Script 2

扩展记号列表，添加 4 个算术运算符：赋值运算符、小括号以及代表标识符的记号。标识符是符号名，如 pi、sin、x 等。

```
enum Etoken
{
  tEnd,
  tError,
  tNumber, // 字面值数字
  tPlus,   // +
  tMult,   // *
  tMinus,  // -
  tDivide, // /
  tLParen, //(
  tRParen, //)
  tAssign, // =
  tIdent   // 标识符(符号名)
};
```

Scanner 现在可以识别符号名字了。所以，除了新方法 GetSymbolName 外，它还有两个数据成员：_lenSymbol（指定当前识别符号名的长度）以及_iSymbol（在 Scanner 的缓冲器中标出名字的开始）。

```
class Scanner
{
public:
  ...
  void GetSymbolName(char * strOut, int & len);
```

```
private:
  ...
  int       _lenSymbol;
  int       _iSymbol;
};
```

扩展 Accept 方法，让它识别附加的算术符号（减号和除号）、两个小括号、等于号以及浮点数与标识符。在扫描器的 switch 语句中给数字列表中添加小数点，因此，扫描器可以识别像.5 这样以小数点开头的数。

库函数 strtod（字符串到双精度）不仅将一个字符串转换成一个浮点数，而且还更改指向不可能为数字一部分的字符的指针。这非常有用，因为它使让我们很容易计算扫描数字后 _iLook 的新值。

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
{
  _token = tNumber;
  char * p;
  _number = std:: strtod(&_buf [_iLook], &p);
  _iLook = p - _buf; // 指针减
  break;
}
```

函数 strtod 有两个输出：已识别数字的值以及第一个未识别字符的指针。

```
double strtod(char const * str, char **ppEnd);
```

函数如何才能有多个输出呢？办法是作为参数传递一个指向函数所修改值的引用或指针。在这里，附加的输出是字符的指针。我们必须给此指针传递一个引用或指针（因为 strtod 是一个来自标准 C 库的函数，它使用的是指针而不是引用）。让我们看看发生的一切，一步步进行探讨。首先定义将被 strtod 修改的变量。此变量是一个指向字符的指针：

```
char * p;
```

请注意，我们不必对它进行初始化，它将在后面的调用中被覆盖。下一步，我们传递此变量的地址给 strtod：

```
_number = std::strtod(&_buf [_iLook], &p);
```

此函数需要一个指向某字符的指针的指针：

```
double strtod(char const * str, char **ppEnd);
```

通过反引用指针的指针，strtod 可以覆盖此指针的值。例如，它可以：

```
*ppEnd = pCurrent;
```

这将使原来的 p 指向 pCurrent 所指的对象（如图 5-2 所示）。

在 C++中，我们可以给指针传递引用（倒不是因为它阅读性更强）。

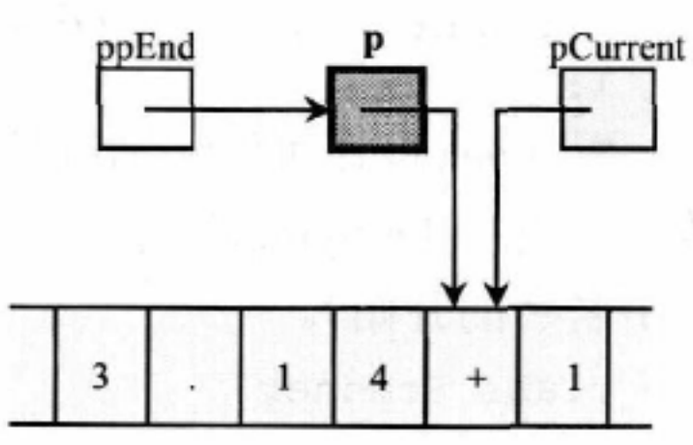


图 5-2　ppEnd 包含指针 p 的地址。指针 p 于是可以通过 ppEnd 来改变

```
char *& pEnd
```

通过引用传递 char*或 int 之类的简单类型是否会得到可读性更强的代码无从知晓。请看下面代码：

```
char * p;
_number = StrToDouble2(&_buf [iLook], p);
```

看起来像给函数传递了一个没有初始化的变量。只有查看 StrToDouble 2 的声明后才能知道 p 通过引用来传递：

```
double StrToDouble2(char const * str, char *& rpEnd)

{
  ...
  rpEnd = pCurrent;
  ...
}
```

在调用函数之前，至少要查看它的声明，这无疑是一个好的编程习惯，但是有的程序员会认为，当你阅读别人的代码时无需这样做。那么，如果你不知道 StrToDouble2 的作用，你又如何理解其代码呢？如果用一条注释语句立即解释其作用，效果就会不同。

```
char * p; // p 将被 StrToDouble2 初始化
_number = StrToDouble2(&_buf [_iLook], p);
```

当你定义一个没有立即初始化的变量时，应该包含一条注释加以提醒。否则，代码的读者会怀疑这是一个 bug。

考虑以上所有因素，我推荐你继续使用引用来传递简单类型及用户自定义类型。

当然，如果 strtod 不是由人工优化编译器编写，代码可能就是下面的样子：

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
{
  _token = tNumber;
  _number = StrToDouble(_buf, _iLook); // 更新_iLook
  break;
}
```

StrToDouble 的声明如下：

```
double StrToDouble(char const * pBuf, int & iCurrent);
```

它将从 pBuf [iCurrent]开始把字符串转换成双精度数，而且让 iCurrent 越过此数字的结尾。

返回到 Scanner::Accept。在庞大的 switch 语句的缺省语句中识别标识符。识别原理是：如果字符不是一个数字、一个小数点或一个运算符，则它必定是一个标识符或一个错误。在计算器中，我们要求标识符以大写或小写字母开头，或者带有下划线。顺便指出，这与 C++标识符的要求完全一致。我们使用 isalpha 函数（实际上是一个宏）来检查字母表中的字母。我们（以及 C++）允许在标识符中同时使用数字。宏 isalnum 检查字符是否包含字母数字。

例如标识符 i、pEnd、_token、__iscsymf、Istop4digits、SERIOUS_ERROR_1 等。

```
    default:
      if(std::isalpha(_buf [_iLook])
          || _buf [_iLook] == '_')
      {
          _token = tIdent;
          _iSymbol = _iLook;

          int cLook; // 在 do 循环中初始化
          do {
             ++_iLook;
             cLook = _buf [_iLook];
          } while(std::isalnum(cLook)|| cLook == '_');

        _lenSymbol = _iLook - _iSymbol;
        if(_lenSymbol >= maxSymLen)
           _lenSymbol = maxSymLen - 1;
}
else
  _token = tError;
break;
```

为了方便程序员，我们限制符号的大小为 maxSymLen。记住，我们现在仍然处于业余程序员的水平，还没有必要复杂化![1]

一旦 Scanner 识别出一个标识符，它必须提供它的名字以供程序的其他部分使用。为了读取符号名，我们调用下面的方法：

```
void Scanner::GetSymbolName(char * strOut, int & len)
{
  assert(len >= maxSymLen);
  assert(_lenSymbol <= maxSymLen);
  std::strncpy(strOut, &_buf [_iSymbol], _lenSymbol);
  strOut [_lenSymbol] = '\0';
  len = _lenSymbol;
}
```

注意，我们必须复制此字符串，因为缓冲器中的原始字符中为非 null 终止。我们将此字符串复制到长度为 len 的调用者的缓冲器 strOut 中。我们通过调用函数 strncpy（string-n-copy 之意，其中 n 的意思是所复制字符的最大数）来实现。长度是 GetSymbolName 方法的 in/out 参数。它必须由调用者初始化为缓冲器 strOut 的大小。在 GetSymbolName 返回后，它的值反映了所复制字符串的实际长度。我们如何知道缓冲器是否足够大呢？我们将它作为契约的一部分——参见断言。

[1] 如果你担心如何来强加这些限制及处理相应的错误，那你的感觉是对的，这种担心并不多余!我们将在本书的第二部分“技术”中解决这一问题，那时，我们就不再是一个业余程序员了。现在，我们暂时无能为力。

方法 GetSymbolName 是一个在对象间传递数据缓冲器的更一般模式的例子。有三种主要方案：调用者的固定缓冲器、调用者分配的缓冲器及被调用者分配的缓冲器。在这里，缓冲器通过调用者传递，而且它的大小固定。这样允许调用者使用一个本地固定缓冲器——不需要在每次调用此函数时分配和重分配。下面是此调用的 Parser 代码（缓冲器 strSymbol 是一个本地数组）：

```
char strSymbol [maxSymLen];
int lenSym = maxSymLen;
_scanner.GetSymbolName(strSymbol, lenSym);
```

请注意，此方法只可以用于缓冲器定义、最大值合理的情形，或者当数据可以不断通过多次调用读取时。这里，我们总是删减 identifiers 的大小使其在 maxSymLen 范围内。

如果在缓冲器中传递的数据大小没有限制，我们必须根据需要分配缓冲器。在调用者分配缓冲器的情况下，我们有两个选择。最佳情形是，调用者必须能首先询问数据大小、分配适当的缓冲器，然后调用方法来填充此缓冲器。有多种方案可以实现调用者分配缓冲器，调用者分配任意大小的缓冲器，它满足 99%的情况。当数据不适合于此缓冲器时，被调用者返回对应的失败代码并让调用者分配更大的缓冲器。

```
char * pBuf = new char [goodSize];
int len = goodSize;
if(FillBuffer(pBuf, len)== errOverflow)
{
   //很少需要
   delete [] pBuf;
   pBuf = new char [len]; // 用 FillBuffer 更新 len
   FillBuffer(pBuf, len);
}
```

如果不是碰到询问数据大小的开销实太大时，这种优化看起来似乎有点奇怪。例如，你可能会跨网络查询，或通过磁盘访问来得到此大小等。

被调用者分配缓冲器似乎是一种相当简单的方案。因为被调用函数知道它需要多少空间，它因此分配此缓冲器并返回一个指向它的指针。最可能的复杂情况是，当调用者忘记了释放缓冲器时（调用者没有显式分配缓冲器，这是我们必须记住的）所发生的内存泄漏。我们将在第 10 章中探讨如何使用智能指针来阻止此类问题的发生。当被调用者使用了调用者以外的其他内存分配器，或者调用越界使用，例如，远程过程调用（Remote Procedure Call，RPC，程序间相互交谈的一种协议），此时，还会发生其他一些复杂情况。通常在处理必须返回动态数据结构的函数（列表、树等）时，我们让被调用者分配内存。下面是被调用者分配缓冲器的一个简单示例代码：

```
char * pBuf = AcquireData();
// 使用 pBuf
delete pBuf;
```

下面的决策树总结了给调用者传递数据的不同方法：

```
if(max data size well-defined)
```

```
{
  use caller's fixed buffer
}
else if(it's cheap to ask for size)
{
  use caller-allocated buffer
}
else if((caller trusted to free memory
         && caller uses the same allocator
         && no problems with remoting)
     || returning dynamic data structures)
{
  use callee-allocated buffer
}
else
{
  use caller-reallocated buffer
}
```

在本书的第二部分，我们将谈到使被调用者分配缓冲器更具吸引力、更方便的一些有趣的方式。

5.3　符号表

隐式转换、关键字 explicit（显式）。

Script 3

通过少量的修改，我们将重用哈希表和字符串缓冲器代码来实现符号表。但是，我们首先要让哈希表在构建期间能设置其大小。

```
explicit HTable(int size): _size(size)
{
  _aList = new List [size];
}

~HTable()
{
  delete []_aList;
}
```

请注意构造函数前面的关键字 explicit。将它放在这里是为了告诉编译器不要使用此构造函数进行隐式转换。通常情况下，当你定义一个带 *T* 类型的单个参数构造函数时，编译器可以使用它来将 *T* 类型对象转换成构造函数所属类的某个对象。例如，因为哈希表的构造函数带一个 int 类型的参数，编译器将接受如下语句：

```
HTable table(13);
```

```
table = 5;
```

它将整数 5“转换”成一个哈希表，与将整数转换成双精度数一样容易。此类隐式转换有时特别有用，但在其他时候，它会带来意想不到的问题。以上代码起到了一点作用，出现的错误也容易排除。不过，想象一下如下情况：有人错误地将一个 int 传递给一个需要指向哈希表的常量引用的函数。此时，编译器将创建一个临时哈希表，它的大小与整数值相等，而且将它传递给此函数。为了防止此类错误的发生，必须养成声明单参数构造函数为 explicit 的习惯，当然，除非你的确想定义一个隐式转换。

符号表是对前面的字符串表简单修改而成（参见 3.5.3 节的内容）。它使用一个哈希表来映射字符串到短字符串 ID 列表。新字符串保存在字符串缓冲器的第一个可用偏移中，而且赋予第一个可用 ID，即_curId。字符串在缓冲器中的偏移然后保存在一个单独的数组_offStr 中。字符串 ID 兼作此数组的索引。

```
const int idNotFound = -1;

class SymbolTable
{
public:
  explicit SymbolTable(int size);
  ~SymbolTable();
  int ForceAdd(char const * str, int len);
  int Find(char const * str)const;
  char const * GetString(int id)const;
private:
  HTable       _htab;
  int          _maxId;        // _offStr 大小
  int          * _offStr;     // 字符串在缓冲器中的偏移
  int          _curId;
  StringBuffer  _strBuf;
};
```

下面是符号表的构造函数和析构函数。除了现在的字符串缓冲器有一个动态分配的缓冲器外（它的大小在构造函数中被传递），其他方法与前面字符串表的实现代码一样。

```
SymbolTable::SymbolTable(int size)
  : _curId(0),
    _maxId(size),
    _htab(size + 1),
    _strBuf(size * 10)
{
  _offStr = new int [size];
}

SymbolTable::~SymbolTable()
{
  delete []_offStr;
```

}

用于初始化哈希表和字符串缓冲器的值有一点试探性。我们希望哈希表比保存在其中的条目数量稍多一些，也希望标识符的平均大小少于 10 个字符。

5.4　存储

前向声明（Forward declaration）。

Script 4

计算器可以处理符号变量。用户通过起一个名来创建一个变量，然后在算术运算中使用它。每个变量在用于求其他表达式的值前都必须初始化——在赋值表达式中赋值。为了保存用户自定义变量的值，计算器需要有某种类型的“记忆”。我们要创建一个类 Store，它包含内存单元的固定数（称做 size）。每个单元可以保存 double 类型的一个值。单元编号从 0 到 size-1。每个单元可以为两种状态之一——未初始化或已初始化。

```
enum { stNotInit, stInit };
```

符号名字（一个字符串）以及单元号间的联系由符号表进行处理。例如，当用户首先引入一个给定的变量 x，字符串"x"就被添加到符号表中并赋值为整数 3。从此以后，变量 x 的值将保存在 Store 对象中编号为 3 的单元里。

我们还要预初始化符号表并保存一些有用的常量，如 *e*（它是自然对数的底数）、*pi*（圆周率）。我们打算在 Store 的构造函数中完成，所以我们需要传递一个指向符号表的引用。现在出现一个矛盾：我们要将类 Store 的定义放在一个单独的头文件 store.h 中，而类 SymbolTable 的定义位于一个不同的文件 symtab.h 中。当编译器看到 Store 的构造函数声明时：

```
Store(int size, SymbolTable & symTab);
```

它并不知道 SymbolTable 是什么。简单的办法是在 store.h 中包含文件 symtab.h。这样做没有错，只是增加了编译器处理 symtab.h 或其他包含文件时需要处理更多文件的负担。在相当大的项目中，许多头文件间互相包含可能会导致麻烦。而且，如果你正使用一种依赖性检查器，它会认为 symtab.h 的变化需要对直接或间接包含它的所有文件都进行重新编译。特别是，任何包含 store.h 的文件也必须要重新编译。所有这些不必要的工作都必须要完成，因为我们需要让编译器知道 SymbolTable 是一个类的名字。为什么我们不直接告诉它呢？的确应该如此，此类前向声明的语法为：

```
class SymbolTable;
```

只要我们只使用 SymbolTable 的指针或引用，它就会发挥作用。我们就不需要包含 symtab.h 了。

另一方面，如果我们想调用 SymbolTable 的任何一个方法（包括构造函数和析构函数），或者我们想嵌入或继承 SymbolTable，前向声明就不能胜任了。

下面是 Store 的定义：

```
class SymbolTable; // 前向声明

class Store
{
public:
  Store(int size, SymbolTable & symTab);
  ~Store()
  {
    delete []_cell;
    delete []_status;
  }
  bool IsInit(int id)const
  {
    return(id < _size && _status [id] != stNotInit);
  }
  double Value(int id)const
  {
    assert(IsInit(id));
    return _cell [id];
  }
  void SetValue(int id, double val)
  {
    if(id < _size)
    {
      _cell [id] = val;
      _status [id] = stInit;
    }
  }
private:
  int           _size;
  double       *_cell;
  unsigned char *_status;
};
```

Store 包含两个数组：单元数组和状态数组。数组在构造函数中初始化并在析构函数中删除。我们还保存了这些数组的大小（它用于错误检查）。Store 的客户可以检查某单元是否已经初始化，得到保存在那里的值以及设置（和初始化）此值。

为什么我们使用两个数组而不使用双域对象的单个数组呢？我可以定义类 Cell 为：

```
class Cell
{
  // ...
private:
  double        _value;
  unsigned char  _status;
};
```

我们在 Store 中有了一个 Cell 数组，而不是针对值和状态的两个单独数组。

两种方案均各有优点。使用 Cell 提高了代码的可读性和可维护性。另一方面，使用两个单独数组可以更有效地使用空间。为什么呢？这是因为数据的排列方式。

在大部分计算机架构中，如果数据以 32-位或 64-位边界排列，那么从内存中读取数据更容易。所以，除非你指定了其他方式，否则编译器将强迫用户自定义数据结构按此边界排列。特别是，它可以填充数据结构。编译器很可能对每个 Cell 添加填充 3 个（甚至 7 个）字节。这样，内存中 Cell 的大小才能保证为 32（或 64）位的倍数，访问 Cell 数组的速度也就非常快。

只要你处理的是恰当范围内的小数组，你就可以不关心这一点。但是如果你的程序使用了大量的内存，你可能要加以小心了。在实现过程中，我们通过声明数组_status 为字节数组（无符号字符）来尽量减少内存消耗。事实上，使用位数组可以侥幸安然无事。毕竟状态为包含两个值的变量，它可以用一个位来表示。实现一个位数组与众不同，每次访问时都包含显式位运算，但在某些情况下，它可能会节省许多内存。

一般原则是：如果你由于内存的限制期望压缩数据结构，则单独实现可压缩部分将带来更多的灵活性。在这里，我们可以将来作为一个位数组实现_status 而不必更改太多的代码。另一方面，如果你期望可以对数组的每个元素添加更多的较小变量，你最好将它们组合到一个单独的类中。在这里，如果对每个变量需要一些枚举类型，我们可以很容易地将它添加到 Cell 类。

根据不同的情况作出不同的选择正是编程有趣的地方。

Store 的构造函数在源文件 store.cpp 中定义。因为构造函数调用 SymbolTable 的实际方法，所以，此类的前向声明不再足以胜任，我们需要显式在 store.cpp 中包含头 symtab.h。

```
Store::Store(int size, SymbolTable & symTab)
  : _size(size)
{

  _cell = new double [size];
  _status = new unsigned char [size];
  for(int i = 0; i < size; ++i)
    _status [i] = stNotInit;
  //添加预定义常量
  //注意: 如果需要更多，则做更一般性的工作
  std::cout << "e = " << std::exp(1)<< std::endl;
  int id = symTab.ForceAdd("e", 1);
  SetValue(id, std::exp(1));
  std::cout << "pi = " << 2 * std::acos(0.0)
      << std::endl;
  id = symTab.ForceAdd("pi", 2);
  SetValue(id, 2.0 * std::acos(0.0));
}
```

我们在符号表中添加大小为 1 的字符串 e 的映射，然后使用返回的整数在调用 SetValue

时作为一个单元号。初始化 pi 的值时使用了同样的过程。

5.5 函数表

元程序、函数指针、显式数组初始化、显式类初始化。

Script 5

对于一个好的科学计算器，内置函数是必需的。计算器必须能计算平方根、对数、三角函数等等。因为标准 C 库已经实现了大部分基本数学函数，所以我们可以直接使用。我们所要做的是让解析器识别函数调用并调用适当的库函数。唯一需要技巧的地方是完成函数名字间的连接，即解析器识别的字符串与所调用的对应函数之间的连接。

一个途径是通过比较字符串与预定义的函数名列表创建多分支条件，然后在找到匹配后调用对应的函数。

```
if(std::strcmp(string, "sin")== 0)
{
  result = sin(arg);
}
else if ...
...
```

只要内置函数的数量适中，此方案就足以完成任务。假定这不是一个玩具计算器，我们正编写的是一个工业强度型程序，它所处理的内置函数即使不上千，也有成百个。问题是：给定一个字符串后，要对上百个预定义字符串查找匹配。显然，每次都进行上百次字符串比较是不可接受的。

但是，稍安勿躁!我们已经有了一个字符串匹配对象——符号表。因为它是作为一个哈希表来实现的，所以它可以定时执行匹配，而且与表的大小无关。符号表将字符串转换成一个整数。我们可以用内置函数名预填充此表（就像我们用常量填充一样），而且基于整数而不是字符串来分派函数调用。例如，我们可以在符号表的第 0 个位置插入字符串 sin，在第 1 个位置插入 cos，依此类推，接着使用一个 switch 语句来分派调用：

```
case 0:
  result = sin(arg);
  break;
case 1:
  result = cos(arg);
  break;
...
```

使用了一套连续标签（0、1、2 等）的 switch 语句由编译器实现为一个带有常量切换时间并与标签数量无关的跳转表。这看起来是一个完美、定时的方案。

但是我们如何保证 sin 总是对应于 0，cos 总是对应于 1……？我们可以总是以这一特定

的顺序用这些字符串来初始化符号表。毕竟，我们知道符号表分配有从 0 开始的连续索引。但问题解决了吗？实现符号表还有一些奥秘。如果下一个维护程序的人重写此符号表而使用了一个更好的算法，那又如何呢？如果程序员没有分配从 0 开始的连续索引呢？

而且，此代码的可扩展性如何？假如在将来的某个时间，我们想添加更多的内置函数，我们必须对此代码作出哪些修改呢？我们必须：

- 在 switch 语句中添加更多的事例（case）。
- 在符号表中添加更多的字符串。
- 确保为字符串分配了一个 ID，它与对应函数的 case 标签相同。

注意我们刚才所做的事。我们编写一个元程序——为了在计算器中添加新内置函数的一套指令。它们不是机器指令；它们是相对于程序员的指令。事实上，它们不会出现在程序的任何地方，甚至不会以注释形式出现——它们是隐式的。这种不可见元代码增加了程序所隐藏的复杂性。元代码描述的内容是什么？它描述的是扩展程序最可能的实现步骤，以及进行扩展时必须保存的断言。

> 当比较不同的实现代码时，不仅要考虑代码的复杂性，还要考虑元代码的复杂性。

让我们看看，是否可以找到内置函数的更好实现方式。最佳办法是用一些类型的表列出所有的函数（它几乎总是一种改变从代码到数据结构的复杂性的好办法）。添加一个新函数等价于在表中添加一个新项。对应的元程序将包含一条语句：

Add a new entry to the function array（在函数数组中添加新项）。

为了实现数据和可执行代码间的连接，我们需要一个新设备：函数的指针。函数的指针就像一个常规指针一样，但它不是指向数据，而是指向可执行代码。通过使用数据指针，你可以读和修改数据；通过使用函数指针，你可以调用函数。与其他任何指针类型一样，我们必须能够：

- 声明一个函数指针。
- 初始化指针指向一个特定的函数。
- 通过指针调用函数。

例如，我们可以声明一个带有一个双精度数（作为参数）并返回一个双精度数的函数的指针。这样的函数有许多，我们可以初始化指针让它指向其中的任何一个。特别是，我们可以初始化指针指向函数 double sin（double x）。此后，如果我们通过此指针调用函数，则将调用 sin。如果我们初始化指针指向 double cos（double x），我们就会调用 cos。函数指针的优势在于它可以在程序的执行期间指向不同的函数。进行函数调用的程序部分不必知道所调用的函数是什么。

让我们看一些实际的代码，了解相应的语法。首先，让我们声明一个指向带双精度数、返回双精度数的函数的指针 pFun：

```
double(* pFun)(double x);
```

将它与带一个双精度数及返回一个双精度数的函数 sin 的声明相比：

```
double sin(double x);
```

为什么必须将星号和指针放在小括号中？为了区别函数指针与返回一个指针的函数的声明，我们必须这样做。如果没有小括号：

```
double * pFun(double x);
```

它声明了一个带一个双精度数并返回一个指向双精度数的指针的函数。两者天壤之别!

你可以通过包含这类任何函数的声明并用（* pFun）替换函数名来声明此类型函数的指针。当然，你可以使用一个与 pFun 不同的任意名字。

还记得函数 strtod 吗？它的声明如下：

```
double strtod(char const * str, char **ppEnd);
```

指向此类函数的指针声明如下：

```
double(* pStrtod)(char const * str, char **ppEnd);
```

就是如此简单!

因为函数指针的存在，我们不得不扩展阅读声明的规则。我照常开始反向阅读：pStrtod 是一个指针……但当我们碰到小括号时，我们立即进入匹配括号对并改变解析方向。现在我们看到的第一个东西是左括号，它是参数列表的开始。加上括号的参数列表表示我们正处理一个函数，因此，我们继续往下读：……指向带有 str 的函数，str 是一个指向 const char 的指针，还有 ppEnd，它是一个指向某 char 的指针的指针。然后，我们碰到了闭合括号，所以，我们再次返回到左边仔细阅读的地方并继续阅读：……返回双精度数。同样，当我们碰到一个左方括号时，我们知道正在处理一个数组，例如：

```
int(* pArr)[3];
```

读作：←pArr 是一个指针→指向一个 3 元素数组←元素为整数。（箭头标明了阅读方向的变化）让我看一个更复杂的例子：

```
double(*(* pFunArr)[6])(double x);
```

←pFunArr 是一个指针→指向一个 6 元素数组←6 个元素为指针→它指向带有 double x 的函数←函数返回一个 double。最后，请读者练习下面的声明：

```
double(*(* pFunFunFun)(double(* pFun)(double x)))(double x);
```

请不要轻易尝试使用这样的声明!

现在进行第二步：初始化。为了使指针指向一个函数，我们必须带上函数的地址并将它赋值给此指针。碰巧的是函数的名字正是函数的地址。所以，我们可以按如下方式初始化指针：

```
double(* pFun)(double x)= std::sin;
```

或在其后进行赋值：

```
pFun = std::sin;
```

最后，为了通过指针调用函数，我们反引用指针并传递适当的参数：

```
double x = 3.1415;
double y =(* pFun)(x);
```

函数指针可以用作数组、类、指针的指针等更复杂数据结构的基石。为了简化此类数据结构的声明，通常使用类型定义。

所有的内置函数都带有一个 double 类型的参数，而且返回一个也是 double 类型的结果，所以，我们开始通过类型定义实现内置函数表：

```
typedef double(*PtrFun)(double);
```

函数与其字符串名之间的连接通过类 FunctionEntry 建立起来。

```
class  FunctionEntry
{
public:
  PtrFun  pFun;
  char*   strFun;
};
```

你可能想知道为什么我们让此类的所有数据成员成为公有[1]，但没有提供一个构造函数来初始化它们。这是因为我们希望能够显式初始化它们，如：

```
FunctionEntry funEntry = { std::sin, "sin" };
```

在这类初始化中，你可以直接给对象的数据成员赋值，当而且仅当对象不是既有私有数据又有构造函数时，这是可能的。我仍然没有解释为什么有人想使用这种类型的直接初始化，而不是简单地提供一个适当的构造函数。原因如下：

```
const int maxIdFun = 16;
FunctionEntry funArr [maxIdFun] =
{
  std::log,      "log",
  std::log10,    "log10",
  std::exp,      "exp",
  std::sqrt,     "sqrt",
  std::sin,      "sin",
  std::cos,      "cos",
  std::tan,      "tan",
  CoTan,         "cotan",
  std::sinh,     "sinh",
  std::cosh,     "cosh",
  std::tanh,     "tanh",
  std::asin,     "asin",
  std::acos,     "acos",
  std::atan,     "atan",
  0,          ""
};
```

经过一个循环，我们可以初始化整个 FunctionEntries 的数组。这正是我们所希望达到的。请注意，现在添加新的内置函数是多么的容易：使用一个函数指针和一个字符串名，在列

[1] 根据 C 传统，所有数据成员都为公有的类通常声明为 struct。在纯 C++中使用 struct 的唯一优点是不必在起始处书写 public:。

表中的任意两行间只添加一行代码，它就会工作（你同时需要增加 maxIdFun，但如果你忘记了，编译器会提醒你）。如果你想知道有关聚集（如类和数组）显式初始化的更多知识，请参阅 5.9 节。

既然我们已经掌握了预备知识，下面让我们回到原来的问题上来：通过内置函数名对符号表的初始化。这必须在程序的构建阶段完成——计算器只有符号表预先填充了才能使用。我们因此定义一个名字为 FunctionTable 的对象，目的有两个，一个是用名字初始化符号表，另一个是将符号 ID 翻译成相对所有内置函数的函数指针。

```
class SymbolTable;
class FunctionTable
{
public:
  FunctionTable(SymbolTable & symTab,
                FunctionEntry funArr []);
  int Size()const { return _size; }
  PtrFun GetFun(int id)const
  {
    return _pFun [id];
  }
private:
  PtrFun   _pFun [maxIdFun];
  int      _size;
};
```

FunctionTable 的构造函数带有一个对符号表的引用，并用来自 FunctionEntrys 数组 funArr 的字符串进行填充。同时，它将对应函数指针复制到自己的数组中（严格地说，如果我们让函数表直接使用函数数组就可避免复制——这一优化最好留给读者去练习）。

```
FunctionTable::FunctionTable(
    SymbolTable & symTab, FunctionEntry funArr [])
  : _size(0)
{
  for(int i = 0; i < maxIdFun; ++i)
  {
    int len = std::strlen(funArr [i].strFun);
    if(len == 0)
       break;
    _pFun [i] = funArr [i].pFun;
    std::cout << funArr[i].strFun << std::endl;
    int j = symTab.ForceAdd(funArr [i].strFun, len);
    assert(i == j);
    ++_size;
  }
}
```

注意，循环中的断言非常重要。我们要确保符号表将连续索引赋值给函数的假设真正成立。我们希望字符串 log 对应于 ID 0，因为我们将函数 log 的指针保存在偏移 0 处，依此类推。

目前所有的内置函数都为库函数（在标准 C 库中定义），只有一个例外，这就是 cotangent。cotangent 是 tangent 的反函数，所以容易写出我们自己的实现代码。惟一值得注意的地方是处理 0 的反函数——用 0 除导致异常的发生，程序将终止。这是我们在反函数中测试 0 并返回 HUGE_VAL（在<cmath>中定义）的原因。这不完全正确（被 0 除的结果没有定义），但我们暂时这样做：

```
double CoTan(double x)
{
    double y = std::tan(x);
    if(y == 0)
    {
        std::cout << "cotan of " << x << " undefined\n";
        return HUGE_VAL;
    }
    return 1.0 / y;
}
```

注意在 funArr 初始化中的对应代码行：

```
CoTan, "cotan",
```

包含有自定义 cotangent 函数的指针，也就是 CoTan。为了让编译器知道 CoTan 是什么以及如何正确调用它，我在文件 FunTab.cpp 的顶部添加了此函数的声明。声明和定义间的区别在于前者不提供实现代码；它只是指定函数的信号——参数数量、类型及返回类型。

```
double CoTan(double x);
```

函数声明也称为原型（prototype），它总是在后面带一个分号。如果你想调用（或者在这里是给一个指针赋值）一个无原型的函数，编译器将提示你“丢失了原型”。与定义不一样，声明可以放在同一个文件或不同的文件中重复。你也可以将声明放在头文件内。

对象、变量及其他数据结构也可以在定义前声明。不过，因为数据结构的定义并不是必需它的初始化，所以编译器难以将它与声明区分开来。例如，

```
int x;
```

编译器总是将它解释成一个定义。当看到此定义时，编译器将为变量 x 分配存储空间（或全局，或在堆栈上）。如果你重复此定义或将它放在一个头文件中，而此头文件包含于多个实现文件中，你就会得到编译器的错误提示——“符号多重定义”。

所以，数据的声明前面必须加一个关键字 extern。例如，为了告诉编译器在一些实现文件中定义有一个整数变量，名字为 x，你可以按下面方式声明它（不允许初始化）：

```
extern int x;
```

同样，为了告诉其他人有一个数组 funArr 可用于其他文件，我们可以在头 FunTab.h 中添加如下声明：

```
extern FunctionEntry funArr [];
```

小结：为了避免元代码的一些隐藏复杂性，我们遇到了许多困难。在这种情况下，这种努力是值得的，因为我们知道内置函数表是将来需要修改的第一个元素。我们确保了将来的修改简单易行。我们将它们集中在一个地方（funArr 的初始化中），犯过一些编译器可以发

现的小错误（忘记了更新 maxIdFun），在一个重要位置放置了断言（在 FunctionTable 的构造函数中）。函数表能开始工作，但从设计的角度看，它的实现还不能令人完全满意。我们将在后面返回过来再完善它。

5.6 节点

Script 6

现在是添加更多节点类型的时候了（如图 5-3 所示）。

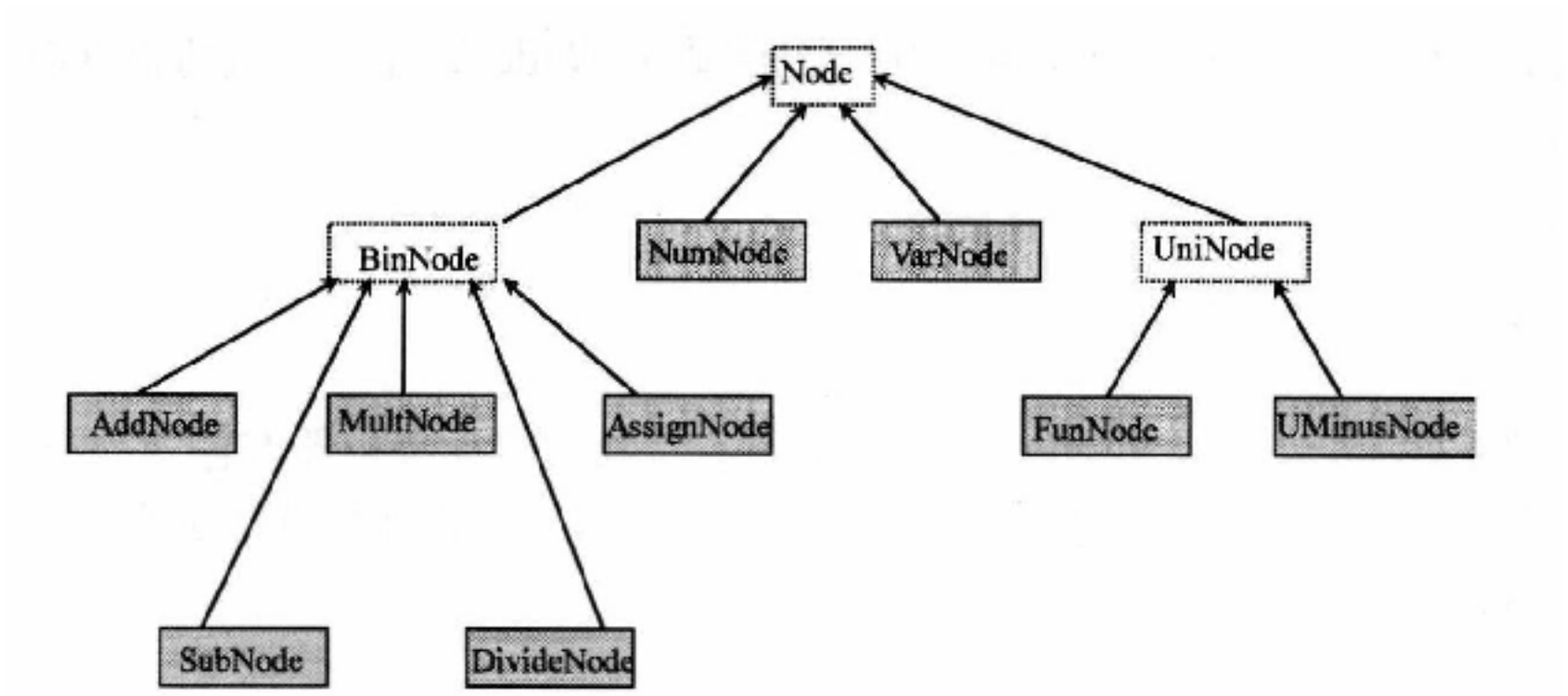


图 5-3 节点类的层次

与内置函数对应的是 FunNode 类型的对象。

```
class FunNode: public UniNode
{
public:
  FunNode(PtrFun pFun, Node * pNode)
    : UniNode(pNode), _pFun(pFun)
  {}
  double Calc()const;
private:
  PtrFun _pFun;
};
```

基类 UniNode 与 BinNode 非常相似，除了它有一个后代而不是两个后代以外（我们将从此类派生 UMinusNode）。FunNode 的 Calc 方法说明了通过指针调用函数的语法。

```
double FunNode::Calc()const
{
  assert(_pFun != 0);
  return(*_pFun)(_pNode->Calc ());
}
```

顺便说一下，通过指针调用函数还有一种语法。你可以简单地将指针看作一个函数名，

如下所示：

```
return _pFun(_pNode->Calc());
```

赋值节点有些技巧。它是一个二元节点，包含对应于左边的后代及右边的赋值。例如，解析下面的代码行：

```
x = 1
```

将得到一个赋值节点，其左节点对应于符号变量 x，右边节点对应于值 1。并不是所有的节点都可以在赋值的左边。例如：

```
2 = 1
```

显然此语句不可接受。我们需要一种方式来判断给定节点是否可以出现在赋值的左边——是否可以为一个左值（lvalue）。在计算器中，唯一的可能左值是符号变量。

为了能验证给定节点是一个左值，我们将给基类 Node 添加一个虚方法 IsLvalue,并提供缺省的实现代码。

```
bool Node::IsLvalue()const
{
  return false;
}
```

覆盖此缺省的唯一节点类型将是符号变量节点。我们将让解析器在创建 AssignNode 前对赋值左边执行 IsLvalue 测试。在赋值节点中，我们只断言解析器完成了此任务。

```
class AssignNode : public BinNode
 {
public:
  AssignNode(Node * pLeft, Node * pRight)
    : BinNode(pLeft, pRight)
  {
    assert(pLeft->IsLvalue());
  }
  double Calc()const;
};
```

Calc 方法通过右边后代计算所得值调用左边后代的 Assign 方法。与缺省的无任何执行代码的缺省实现一起，我们将给基类 Node 添加虚方法 Assign。

```
virtual void Assign(double value){}
```

只有变量节点可以覆盖此实现代码。

已经说过，AssignNode::Calc 的实现很简单明了：

```
double AssignNode::Calc()const
{
  double x = _pRight->Calc();
  _pLeft->Assign(x);
  return x;
}
```

下一步，我们必须定义对应于符号变量的节点。变量的值保存在 Store 对象中。VarNode 为了计算它自己需要访问此对象。但是正如我们所说的，VarNode 也可以用于赋值的左边，

所以，它必须覆盖虚方法 IsLvalue 和 Assign。

```
class Store;

class VarNode: public Node
{
public:
  VarNode(int id, Store & store)
    : _id(id), _store(store){}
  double Calc()const;
  bool IsLvalue()const;
  void Assign(double val);
private:
  const int _id;
  Store & _store;
};

double VarNode::Calc()const
{
  double x = 0.0;
  if(_store.IsInit(_id))
    x = _store.Value(_id);
  else
    std::cout << "Use of uninitialized variable\n";
  return x;
}

void VarNode::Assign(double val)
{
  _store.SetValue(_id, val);
}

bool VarNode::IsLvalue()const
{
  return true;
}
```

解析器还需要几个节点类：对应减法的 SubNode、对应除法的 DivideNode 以及对应一元取反的 UMinusNode。它们的实现非常简单。

5.7 解析器

私有方法。

解析器需要一个重要的改进。让我们从头文件 parse.h 开始。它包含许多前向声明：

```
class Node;
class Scanner;
```

```
class Store;
class FunctionTable;
class SymbolTable;
```

所有这些类都在 parse.h 中提到了，或通过指针或通过引用——所以不再需要包含定义它们的头文件了。

类 Parser 有一个构造函数，它带有指向解析所有来自 Scanner 记号流所需主要对象的引用。它需要 Store 来创建 VarNodes、FunctionTable 来创建 FunNodes，以及用 SymbolTable 来识别变量和函数名。一旦创建了 Parser，我们只可以要求它对在 Scanner 中传递给它的表达式求值。Eval 方法还有一个附加作用是在屏幕上打印此结果（如果你现在想知难而退，我保证你读过本书的下一章后，就会轻松应付了）。

Parser 使用了许多私有方法来解析表达式、项、因子，而且使用了一个方法来对表达式树求值。私有方法是构建代码的有用结构。它们不能从类的外部调用——它们只是用于实现其他（公有或私有）方法的辅助函数。

```
class Parser
{
public:
  Parser(Scanner & scanner,
         Store & store,
         FunctionTable & funTab,
         SymbolTable & symTab);
  ~Parser();
  Status Eval();
private:
  void   Parse();
  Node * Expr();
  Node * Term();
  Node * Factor();
  void   Execute();

  Scanner       & _scanner;
  Node          * _pTree;
  Status          _status;
  Store         & _store;
  FunctionTable & _funTab;
  SymbolTable   & _symTab;
};
```

构造函数初始化 Parser 的所有私有成员变量（我们已经正在查看实现文件 parse.cpp）。

```
Parser::Parser(Scanner & scanner,
               Store & store,
               FunctionTable & funTab,
               SymbolTable & symTab)
: _scanner(scanner),
  _pTree(0),
```

```
  _status(stOk),
  _funTab(funTab),
  _store(store),
  _symTab(symTab)
{}
```

析构函数递归删除解析树（请与 3.5.2 小节中的链表递归析构函数相比较）：

```
Parser::~Parser()
{
  delete _pTree;
}
```

Eval 方法调用私有方法来解析输入并进行计算：

```
Status Parser::Eval()
{
  Parse();
  if(_status == stOk)
    Execute();
  else
    _status = stQuit;
  return _status;
}
```

Execute 调用解析树中顶层节点的 Calc 方法并打印这一（递归）计算结果：

```
void Parser::Execute()
{
  if(_pTree)
  {
    double result = _pTree->Calc();
    std::cout << " " << result << std::endl;
  }
}
```

解析开始时假定一切东西都是表达式。

```
void Parser::Parse()
{
  _pTree = Expr();
}
```

表达式从某一项开始。此项后面为一个绑定了其他项的运算符——加号、减号或赋值号，或者什么也没有。它们对应于 4 种结果（参见 5.1.1 小节中的语法描述）。

```
Expression is Term + Expression
or Term - Expression
or Term = Expression  // 这一项必须为左值
or just Term
```

你可能想起来了，这些形式定义了一种右结合算术运算符。而且不幸的是，这也意味着赋值也是右结合性的，对于如下表达式：

```
1 + x = 2
```

是合法的。它被解析成：

```
1 +(x = 2)
```

我们暂时要容忍这一缺陷的存在。此缺陷将在本书的 6.6 节中进行修正。

根据此语法，在解析某项后，我们必须要求 Scanner 提供下一个记号并测试是否为 tPlus、tMinus 和 tAssign。

识别记号后立即通知 Scanner 接受它。

```
Node * Parser::Expr()
{
  Node * pNode = Term();
  EToken token = _scanner.Token();
  if(token == tPlus)
  {
    _scanner.Accept();
    Node * pRight = Expr();
    pNode = new AddNode(pNode, pRight);
  }
  else if(token == tMinus)
  {
    _scanner.Accept();
    Node * pRight = Expr();
    pNode = new SubNode(pNode, pRight);
  }
  else if(token == tAssign)
  {
    _scanner.Accept();
    Node * pRight = Expr();
    if(pNode->IsLvalue())
    {
      pNode = new AssignNode(pNode, pRight);
    }
    else
    {
      _status = stError;
      delete pNode;
      pNode = Expr();
    }
  }
  return pNode;
}
```

解析器中的错误恢复带有一些试探性。例如，当我们发现赋值左边并非一个左值，我们将它舍弃并求右边表达式的值。在更复杂的解析器中，在发现第一个错误后继续解析有着一定的意义，要是能对用户报告更多的错误就更有利于用户的修改。我们将在本书的第二部分对错误处理介绍更多的内容。

我们通过 Term 进行相似方式的处理，过程如下：

```
Term is Factor * Term
or Factor / Term
or just Factor
```

方法 Term 应该如下所示：

```
Node * Parser::Term()
{
  Node * pNode = Factor();
  if(_scanner.Token()== tMult)
  {
    _scanner.Accept();
    Node * pRight = Term();
    pNode = new MultNode(pNode, pRight);
  }
  else if(_scanner.Token()== tDivide)
  {
    _scanner.Accept();
    Node * pRight = Term();
    pNode = new DivideNode(pNode, pRight);
  }
  return pNode;
}
```

Factor 依次可以为以下之一：

```
Factor is(Expression)// 括号中的表达式
or Number // 字面浮点数
or Identifier(Expression)// 函数调用
or Identifier // 符号变量
or - Factor // 一元减
Node * Parser::Factor()
{
  Node * pNode;
  EToken token = _scanner.Token();

  if(token == tLParen)
  {
    _scanner.Accept(); // 接受 '('
    pNode = Expr();
    if(_scanner.Token()!= tRParen)
      _status = stError;
    else
      _scanner.Accept(); // 接受 ')'
  }
  else if(token == tNumber)
  {
    pNode = new NumNode(_scanner.Number());
```

```
    _scanner.Accept();
  }
  else if(token == tIdent)
  {
    char strSymbol [maxSymLen];
    int lenSym = maxSymLen;
    // 把符号复制到 strSymbol 中
    _scanner.GetSymbolName(strSymbol, lenSym);
    int id = _symTab.Find(strSymbol);
    _scanner.Accept();
    if(_scanner.Token()== tLParen)// 函数调用
    {
      _scanner.Accept(); // 接受 '('
      pNode = Expr();
      if(_scanner.Token()== tRParen)
        _scanner.Accept(); // 接受 ')'
      else
        _status = stError;
      if(id != idNotFound && id < _funTab.Size())
      {
        pNode = new FunNode(
          _funTab.GetFun(id), pNode);
        }
        else
        {
          std::cout << "Unknown function \"";
          std::cout << strSymbol << "\"\n";
        }
      }
      else
      {
        if(id == idNotFound)
          id = _symTab.ForceAdd(strSymbol, lenSym);
        pNode = new VarNode(id, _store);
      }
    }
    else if(token == tMinus)// 一元减
    {
      _scanner.Accept(); // 接受减
      pNode = new UMinusNode(Factor());
    }
    else
    {
      _scanner.Accept();
      _status = stError;
      pNode = 0;
```

```
  }
  return pNode;
}
```

为了看解析器如何工作，让我们探讨一个简单的例子。例如用户输入

```
x = 1
```

首先创建包含输入字符串的扫描器。它扫描第一个记号并判断它是一个标识符 tIdent。下一步，调用解析器来求表达式的值（通过 Eval）。通过假定它是一个表达式（实际上就是）来开始解析。

```
_pTree = Expr();
```

表达式必须以一个项开始，因而 Expr 调用 Term。

```
Node * pNode = Term()
```

另一方面，Term 期望有一个 Factor。

```
Node * pNode = Factor();
```

最后，Factor 看到了第一个记号

```
EToken token = _scanner.Token();
```

而且将它与 tLParen、tNumber、tIdent 和 tMinus 进行比较。在这里，第一个记号是一个标识符 tIdent。它然后要求扫描器提供符号的名字（名字为 x），而且在符号表中搜索它。现在通过第一个记号来实现 factor，因此它告诉扫描器接受此记号。

```
// 将符号复制到 strSymbol
_scanner.GetSymbolName(strSymbol, lenSym);
int id = _symTab.Find(strSymbol, lenSym);
_scanner.Accept();
```

扫描器扫描下一个记号，它是赋值运算符 tAssign。解析器查找此记号来检查它是否为左括号——左括号表示一个函数调用。因为它不是一个左括号，解析器使用符号表中发现的符号 ID 创建一个 VarNode。如果这是一个不存在 ID 的新符号，解析器将它添加到符号表并创建一个新 ID。

```
if(id == idNotFound)
  id = _symTab.ForceAdd(strSymbol, lenSym);
pNode = new VarNode(id, _store);
```

Factor 完成后返回一个指向刚才所创建节点的指针。现在再回到 Term。Term 有一个来自 Factor 的节点，所以，它现在检查下一个记号是否为 tMult 或 tDivide。因为它不是，所以 Term 返回同样的节点。现在回到 Expr。它再一次检查此记号：它是 tPlus、tMinus 或 tAssign？它是 tAssign！因此 Expr 接受此记号（扫描器将它本身放在值为 1 的 tNumber 位置）。目前为止，解析器看到一个后面带有等于号的节点。一个等于号后面可以为一个任意表达式，所以，Expr 现在可以调用它本身来解析余下的输入，即“1”。

```
_scanner.Accept();
Node* pRight = Expr();
```

Expr 再一次经过 Term 和 Factor 创建一个 NumNode。

```
pNode = new NumNode(_scanner.Number());
```

```
_scanner.Accept();
```

返回到拥有新节点 pRight 的 Term 和 Expr。解析器现在看到一个节点 pNode，后面是等于号和另一个节点 pRight。它看起来非常像一个赋值语句，但解析器不知道 pNode 是否表示一个左值。在这里，VarNode 节点就是这样。一个赋值节点被创建：

```
if(pNode->IsLvalue())
{
  pNode = new AssignNode(pNode, pRight);
}
```

Expr 给 Parse 返回这一 AssignNode。解析树现在如图 5-4 所示。

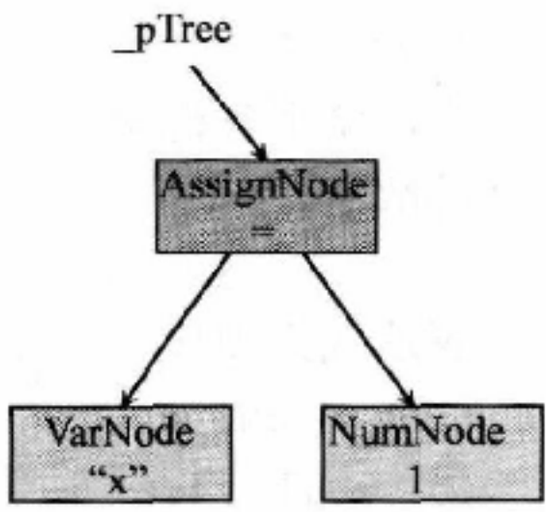


图 5-4　表达式 x=1 的解析树

因为解析成功，Eval 调用 Execute，Execute 又调用 _pTree->Calc。AssignNode 的虚方法 Calc 计算右节点的值——NumNode 返回 1——并通过此值调用 VarNode 的 Assign 方法。VarNode 可以访问 Store 并在对应于 x 的位置保存值 1。

5.8　主函数（main）

main 函数是整个程序的源头。它的最后是一个迭代，整个结构没有太多的改变——除了另外添加了函数表、符号表以及 Store 之外。

```
const int maxBuf = 100;
const int maxSymbols = 40;
void main()
{
  // 注意所有的本地对象
  // 应当有一个顶层对象，即 Calculator
  // 重新开始
  char buf [maxBuf];
  Status status;
  SymbolTable symTab(maxSymbols);
  FunctionTable funTab(symTab, funArr);
  Store store(maxSymbols, symTab);
  Do
  {
     std::cout << "> ";  // 提示
     std::cin.getline(buf, maxBuf);
     Scanner scanner(buf);
     Parser parser(scanner, store, funTab, symTab);
     status = parser.Eval();
  } while(status != stQuit);

}
```

请注意，对于每一行输入，我们创建了一个新的扫描器和新的解析器。不过，要保持同样的符号表、函数表及存储。这一点很重要，因为我们希望只要程序被激活，就记住赋值给变量的这一个值。解析器的析构函数在评估每一行输入后被调用，这一调用对释放解析树起着重要作用。



Script 7

让我们再看一下此项目的结构（参见表 5-1）。你可以看到大部分类都有自己的文件对——头文件及实现文件。有些文件包含多个类。紧密相关的一组类及数据结构有时组合在一起。main 函数通常单独成为一个文件，它的名字或为 main.cpp，或借用应用程序的名字，例如我们所举的这一个例子。

我们将在本书第二部分返回到此项目，探讨如何使它成为一个专业、“工业强度”型程序。但在这之前，我将补充介绍一些在后面内容中有用的概念。

表 5-1　　项目文件及其内容

文　　件	内　　容
Calc.cpp	main
Parser.h, Parser.cpp	Parser
Scan.h, Scan.cpp	EToken, Scanner
FunTab.h, FunTab.cpp	FunctionEntry, funArr, FunctionTable
SymTab.h, SymTab.cpp	SymbolTable
StrBuf.h	StringBuffer
List.h, List.cpp	Link, List
HTab.h, HTab.cpp	HTable
Store.h, Store.cpp	Store
Tree.h, Tree.cpp	Node 及其子代

5.9 聚集初始化

显式初始化类和数组。

与使用字面字符串显式初始化一个字符的数组一样：

```
char const string [ ] = "Literal String";
```

你可以也按同样方式初始化其他聚集数据结构，显式初始化——类和数组。当且仅当类的所有数据成员为公有，而且没有对应的构造函数时，给定类的对象可以显式初始化。这些

公有数据成员必须同时显式可初始化（有一点递归的意味）。例如，对于以下代码：

```
class Initializable
{
public:
  // 没有构造函数
  int         _val;
  char const * _string;
  Foo        * _pFoo;
};
```

你可以定义此类的一个实例并立即初始化它的所有成员：

```
Foo foo;
Initializable init = { 1, "Literal String", &foo };
```

因为 Initializable 可以初始化，所以，你可以将它用作另一个可初始化类的数据成员。

```
class BigInitializable
{
public:
  Initializable _init;
  double        _pi;
};

BigInitializable big =
  {{1, "Literal String", &foo }, 3.14 };
```

正如你所见到的，你可以在大括号中放置大括号来实现嵌套初始化。

你也可以显式初始化对象的某个数组。它们可以为简单或聚集类型，甚至可以为对象的数组的数组。下面是几个例子。

```
char const string [] = { 'A', 'B', 'C', '\0' };
```

它等价于简写式：

```
char const string [] = "ABC";
```

下面是另一个例子：

```
Initializable init [2] = {
     { 1, "Literal String", &fool },
     { 2, "Another String", &foo2 } };
```

我们在 FunctionEntry 对象的数组的初始化中使用了这一方法。

如果数组中的对象有单参数构造函数，你可以在初始化器列表中指定这些参数。例如：

```
CelestialBody solarSystem [] =
  { 0.33, 4.87, 5.98, 0.64, 1900, 569, 87, 103, 0.66 };
```

其中，行星的质量以单位 10^{24}kg 给出。如果数组 solarSystem 动态分配（使用 new），你就不能使用此方式的初始化。

练习

1. 给计算器添加两个新的内置函数：sqr 和 cube。Sqr 求参数的平方根，cube 求它的立

方根（上升到三次方）。

2. 给计算器添加对一元加的识别。对扫描器和解析器进行必要的修改。添加一个新节点 UPlusNode。此计算器必须能正确处理如下表达式：

```
x = +2
2 * + 7
1 /(+1 - 2)
```

3. 根据下面的描述，给计算器添加幂：

```
Factor 是 SimpleFactor ^ Factor // a ^ b(a 的 b 次幂)
  或 SimpleFactor

SimpleFactor 是(Expression)// 括号中的表达式
  或 Number  // 字面浮点数
  或 Identifier(Expression)// 函数调用
  或 Identifier// 符号变量
  或 - Factor // 一元减
```

4. 在解析树中给所有的节点添加虚方法 Print。当树的根调用 Print()时，整个树必须按一种可读形式显示。使用不同的缩进（在每行开头打印不同数量的空格）来区别树的不同级别。例如：

```
void AddNode::Print(int indent)const
{
  _pLeft->Print(indent + 2);
  Indent(indent);
  std::cout << "+" << std::endl;
  _pRight->Print(indent + 2);
  std::cout << std::endl;
}
```

其中 Indent 打印空格的数量与参数的值相等。

5. 一些内置函数的导数分别是：

```
sin(x)-> cos(x)
cos(x)-> -sin(x)
exp(x)-> exp(x)
log(x)-> 1/x
sqrt(x)-> 1/(2 * sqrt(x))
```

和的导数是导数的和；积的导数计算公式为：

```
(f(x)* g(x))' = f'(x)* g(x)+ f(x)* g'(x)
```

其中撇号表示导数。商的导数计算公式为：

```
f(x)/ g(x))′ =(f′(x)* g(x)- f(x)* g′(x))/(g(x)* g(x))
```

而且复函数的导数计算公式为：

```
(f(g(x))' = g'(x)* f'(g(x))
```

重新编写计算器，根据以上公式通过转换解析树派生符号导数。确保此过程无内存泄漏（也就是说，你必须删除所分配的一切）。

5.10　过程化编程

到目前为止，除了一些库函数外，我们已经使用和实现的函数几乎无一例外是与不同类关联的成员函数。这不是 C++的传统教授方式。大部分 C++书籍都以讲解与任何类无关联的所谓自由函数开始。实际上，类通常在较后的章节中介绍，最好在介绍了 C++的 C 子集之后讲解。

我选择一种不同的讲解途径，是因为我发现与其他方式相比，这种方式更容易从对象过渡到过程。事实上，似乎过程编程更自然。我们倾向于用烹饪法的方式来思考——先做这个、再做那个，再把其他事情重复几次，如此等等。按对象的交互作用来思考问题要难得多，因为它需要进行抽象。

拥有过程编程背景的程序员，尤其是大部分 C 程序员，他们发现面向对象编程不仅困难，而且笨拙、不自然且没必要。即使他们在实现抽象数据类型时应用了面向对象编程，但在解决新问题时他们并不情愿使用它。这是因为他们已经借助于一些真正的基本数据结构通过过程化步骤找到了解决方案。当设计一个计算器时，他们会立即明白所需的几个过程：使用递归函数调用解析、用一个 switch 语句来识别记号以及可能用一个哈希表来保存标识符，再在若干地方加上几个数组。

我没有说过程编程天生就不好，而面向对象编程总是好的。但我相信从面向对象角度开始设计和实现程序代码是更好的选择，而且最后回到过程编程上（毕竟在某些时候，你必须实现方法，而它们最终是过程）。

尝试使用其他的方式——从过程化方案开始并将数据结构看成是以后的应用，这样能产生美观的结构化代码但数据结构凌乱、不可管理。

C++是一种非常折衷的语言。它对过程化编程的支持良好（根据你的评判标准，你也可以说支持得不好），同时对面向对象编程的支持同样优秀（而且同时应用了模板、一般化编程技术）。由于已经学过了面向对象编程的知识，你不必为学习过程化编程而花额外的气力了。让我回顾一些自由函数的知识，并与方法进行比较。

一个自由函数在没有对象的情况下调用。它仅能访问形式参数、本地变量及全局对象。而一个（非静态）成员函数总是在针对特定的对象被里调用，而且可以访问所有对象的成员和方法。它还可以访问称做“this”的指向此对象的指针。当编写下面这样的代码时，访问“this”可以发挥作用：

```
_next = this;
_prev = this;
```

这是我们在第 3 章练习 9 中见过的代码。

在 C++中，自由函数与成员函数一样，它可以定义为内联（它们的定义前置关键字 inline），可以通过参数的数量和类型来重载，而且对于尾参数可以有缺省值。自由函数不能定义为 const，因为这样做没有意义。你已经知道，可以得到一个自由函数的地址并将它赋值给一个

函数的指针。在C++中，成员的指针也允许如此，但它们的语法和用法稍微复杂一些——我们将在后面讨论这一点。

顺便说一下，如果你想组合使用C++代码与一些现有的C代码，你必须告诉编译器，你即将调用的函数是用C写成的。这是因为C有一个不同的调用约定——它将参数按不同顺序放在堆栈中。通知编译器的办法是，在C函数声明的前面放置extern "C"。

同样，如果希望能从C调用自己的自由函数，你必须将它声明为extern "C"：

```
extern "C" void called_from_C(int x, int y);
```

5.11 运算符重载

你几乎可以用C++中的内置整型和浮点类型进行任何种类的算术运算。不过，这还不够。过去的工程师信赖FORTRAN语言，它具有复数内置类型。在许多工程应用中，尤其是在电子学中，如果不使用复数将无法进行真正的有效计算。

C++语言没有内置复数类型。它也不支持矩阵和矢量计算。难道这就意味着工程师和科学家应该忠于FORTRAN吗？绝对不是!显然，在C++中你可以定义新的对象类，所以，定义一个复数易如反掌。如何进行复数的加、减、乘等运算？你可以定义类complex的适当方法来完成。符号运算的方便性又如何呢？在FORTRAN中，你只需在两个复数之间使用一个加号就可以将它们相加。在C++中也没问题!请使用运算符重载。

如下表达式中：

```
double delta = 5 * 5 - 4 * 3.2 * 0.1;
```

你看到了若干算术运算符：等号、乘号以及减号。这意味着编译器能准确理解它们的意义。它知道如何来相乘或相减整数或浮点数。但如果你想让编译器知道如何来乘或减一些自定义类的对象，你必须重载对应的运算符。运算符重载的语法需要时间来适应，但它们的使用却与普通运算符一样（这一点将在稍后内容中讲解）。在两个复数变量间放置一个乘号，编译器找到复数乘法的定义后就可以应用它完成运算。

顺便提及，可以在标准C++库中方便定义一个复数类型。

等号也是一个运算符。与C++中的大部分运算符一样，它也可以重载。但它的意义超越了一般的算术运算。事实上，如果你不将它应用在特殊的场合，你可以应用它给同一个类中的另一个对象赋任意对象，办法是在它们之间放置一个等于号。没错，你可以赋值符号表，如下所示：

```
SymbolTable symTab1(100);
SymbolTable symTab2(200);
symTab1 = symTab2;
```

上面代码中的赋值有意义吗？不，此赋值大部分情况下肯定出错，而且它将会导致内存管理上一个非常棘手的问题。所以，即使你不准备重载标准算术运算符，你也应该学习赋值运算符的一些知识，包括何时以及为什么要重载它。因此引出了一个重要的概念——按值传

递与值语义。

5.12　按值传递

拷贝构造函数、重载赋值运算符、缺省拷贝构造函数及运算符=、按值返回、按值传递、隐式类型转换。

到目前为止，我们在方法间相互传递对象时已经使用了引用或指针。例如，在下面的代码中：

```
Parser parser(scanner, store, funTab, symTab);
```

Parser 构造函数的所有参数——scanner、store、funTab 及 symTab——都按引用进行传递。因为我们已经见过如下声明（编译器同样可以看到此声明）：

```
Parser(Scanner & scanner,
       Store & store,
       FunctionTable & funTab,
       SymbolTable & symTab);
```

当我们构建解析器时，我们不会复制符号表。我们只是访问一个现有的符号表。如果我们进行一个私有化的复制，我们就不能看到解析器对它所作的更改。例如，解析器可能给符号表添加了一个新变量。我们希望符号表记住此变量，甚至在当前解析器销毁后仍然能记住。对于 Store 同样如此——它必须记住调用解析器后赋予符号变量的值。

对于扫描器的情况又是怎样的呢？我们并不真正关心解析器是否对它进行了私有化的复制，我们也不关心解析器将对函数表做什么。我们在这里所关心的是它的操作性能。创建一个庞大对象的拷贝可能会耗费大量时间。

但是设想一下，如果我们不关心它的操作性能。下面的代码将如何工作？

```
Parser(Scanner scanner,
       Store & store,
       FunctionTable funTab,
       SymbolTable & symTab);
```

注意 Scanner 和 FunctionTable 后面的&。我们要告诉编译器的是，当调用者创建一个解析器并传递一个扫描器给它时，对此扫描器进行临时复制，让解析器的构造函数在此拷贝上操作。

> 毕竟，这是内置类型的传递方式。当你调用一个需要一个整数的方法时，在此方法内使用的是此整数的拷贝。你可以在此修改此拷贝，但你决不能更改原始值。只有当你显式请求此方法接受一个对整数的引用时，你才能更改原始值。

有许多原因导致此途径的工作方式并不像我们在 Parser 例子中所期望的一样，除非我们对此代码作一些修改。首先，扫描器（以及函数表）的所有临时拷贝在执行完解析器的构造函数后立即消失。解析器将在它的成员变量中保存一个对它的引用，但这是无用的。在构建

结束之后，此引用将指向一个并不存在的扫描器拷贝，这样并不好。

如果我们决定给解析器传递扫描器的拷贝，我们还应该在解析器中保存此扫描器的一个拷贝。实现的办法是忽略其中的&。

```
class Parser
{
  ...
private:
  Scanner       _scanner;
  Node        * _pTree;
  Status        _status;
  Store       & _store;
  FunctionTable  _funTab;
  SymbolTable  & _symTab;
};
```

但在构造函数内实际发生了什么呢？既然参数 scanner、成员变量_scanner 都不是引用，那如何用 scanner 来初始化_scanner 呢？其语法简单，但容易误解。

```
Parser::Parser(Scanner scanner,
    Store & store,
    FunctionTable funTab,
    SymbolTable & symTab)
: _scanner(scanner),
  _pTree(0),
  _status(stOk),
  _funTab(funTab),
  _store(store),
  _symTab(symTab)
{
}
```

在后台所做的是调用了 Scanner 的拷贝构造函数。一个拷贝构造函数是一个带有对同一个类中某对象的常量引用并且复制了它的函数。在这里，对应的构造函数声明为：

```
Scanner::Scanner(Scanner const & scanner);
```

请注意! Scanner 并没有此信号的构造函数，为什么编译器不拒绝呢？而我们试图调用一个未定义成员函数时，编译器总是发出错误警告呀？如果你没有为给定类显式声明一个拷贝构造函数，编译器将会为你自动创建一个，这一答案出人意料。

小心缺省拷贝构造函数带来的影响!

由编译器生成的拷贝构造函数很可能是错误的![1]毕竟对于不能说话的编译器，它对复制用户自定义类又能知道多少呢？的确，它已经尽力了。它尽力完成了以下任务：

[1] 特别是对于那些具有特殊构造函数（如分配某些对象）以及特殊析构函数（如释放在构造函数中分配的对象）的类，由编译器生成的拷贝构造函数几乎总是错误的。

- 对内置类型的所有数据成员进行位复制。
- 对用户自定义嵌入对象调用各自的拷贝构造函数。

但是另外一种情况是，任何时候只要编译器碰到一个指针，它就对它进行复制；编译器不会创建指针所指对象的拷贝。这样做适宜与否，只有类的编程者才知道答案。

这种操作就叫浅层复制（shallow copy），与之相反的是深层复制（deep copy），它能跟踪所有的指针并复制它们所指的所有对象。当指针所指数据结构容易在对象的多个实例间共享时，使用浅层复制是适宜的。

举例来说，请考虑我们浅层复制一个解析树的顶层节点时的情况。如果顶层节点有子代，浅层复制不会克隆此子节点。我们最后有两个顶层节点，都指向同一个子节点。当调用了其中一个顶层节点的析构函数时就会发生问题，它会立即删除它的子代。那么，第二个顶层节点现在指向什么呢？指向一堆垃圾!当它试图访问子代时，它将重新访问回收过的内存得到异常结果。即使它不做任何事，最终也要调用它的析构函数。析构函数将试图删除已经被第一个顶层节点删除的同一个子代。后果是什么？内存崩溃!

请等一下，还有更多可能带来意想不到结果的地方!不仅 C++暗中提供一个拷贝构造函数，而且它还提供一个它认为方便的缺省赋值运算符。

小心缺省赋值带来的不利影响!

下面代码完全合法：

```
SymbolTable symTab1(100);
SymbolTable symTab2(200);
// ...
symTab1 = symTab2;
```

它不仅将 symTab2 浅层复制到 symTab1，还销毁 symTab1 中的已有内容。symTab1 中分配的所有内存都丢失，永远不会被回收，相反，在 symTab2 中分配的内存将被删除两次!这就是此操作的所得与所失!

为什么 C++要不知不觉地提供这些未经允许的可恶服务呢？如果你继续阅读本书，你就会找到答案——这是为了与 C 兼容!在 C 中，你不能定义拷贝构造函数，因为那里没有任何构造函数。所以，每次想要复制比 int 更复杂的东西时，你必须编写特殊的函数或宏。因此，本着减轻程序员劳动强度的传统精神，C 提供了此后台复制所有用户自定义结构的功能。在 C 中，它的影响并不大——C 结构只是原始的数据。它没有数据隐藏、方法或继承。另外，在 C 中也没有任何引用，所以，不太可能因为忘记了使用一个&而非有意地复制一个结构。但是在 C++中，情况完全不同。请注意——C++中的许多错误都是因为&的丢失。

但还有需要注意的地方!C++不仅提供了一个免费拷贝构造函数以及一个免费的赋值，它还在后台不知不觉地使用它们从函数返回对象。下面是原来基于堆栈计算器的代码片断，其中只有少量的修改。你能找出这些修改之处吗？

```
class Calculator
{
public:
  int Execute(Input & input);
  IStack const GetStack()const {return _stack;}
private:
  int Calculate(int n1, int n2, int token)const;

  IStack _stack;
};
```

其中，我忽略了 Calculator::GetStack 返回类型中的&，现在 IStack 按值返回。让我们详细看看在这一转换中发生的一切。同样，我们暂时假定编译器没有进行任何优化。特别是让 GetStack 的定义不一致，这样就必须执行常规函数调用。

```
IStack const Calculator::GetStack()const
{
  return _stack;
}
//...
IStack const stk = calc.GetStack();  // <- 按值!
```

按值返回一个对象的过程包括两个步骤：拷贝构造函数及赋值。

- 首先，在执行对 GetStack 的调用之前，编译器预先在堆栈上分配一些暂存空间（我在这里所指的是内部调用堆栈，其中函数参数与本地变量在运行时有效）。然后将此空间传递到被调用的函数。为了将_stack 的值复制到暂存空间，GetStack 要调用 Istack 的拷贝构造函数。
- 从 GetStack 返回后，调用赋值运算符从暂存空间将此值复制到本地变量 stk。[1]

初看之下非常复杂，但如果你仔细加以考虑就知道，这的确是按值返回一个对象的简单易行的办法，特别是在程序员既要定义一个拷贝构造函数，又要重载对应赋值运算符时。

对于一些类，编译器不能生成缺省的赋值。这些类包含有不能在构造函数前导外部赋值的数据成员。常量成员（包括常量指针）以及引用就是这样。例如，如果你试图给一个解析器赋值另一个解析器，如下所示：

```
parser1 = parser;
```

编译器就会发出错误提示“没有为类 Parser 定义的赋值运算符”。这是因为它不能重新初始化 parser1 内 Scanner、Store、FunctionTable 以及 SymbolTable 的所有引用。一旦某个引用引用了某事物，你就不能再让它引用其他事物。但是请注意，这些限制并不适用于拷贝构造函数。下面的代码：

```
Parser parser1(parser);
```

能被编译。parser1 中的引用将被正确初始成与 parser 中的引用指向同一个对象。不过，

[1] 优秀的编译器能优化此过程，尤其针对当返回对象构建于返回语句的适当地方时。这称做“返回值优化”。

因为值语义既需要构造函数又需要赋值，以此形式存在的类 Parser 不能按值传递。

顺便提及，对以上代码作少量修改后：

```
Parser parser1 = parser;
```

也能被编译，因为此处的赋值被理解成一个构造函数调用的简写。

5.13　值语义

让我们对以上混乱过程进行剖析。我们已经学过了 C++的几个特征，如果单独使用它们，可能会使代码适得其反。下面讨论它们应用在正确场合的作用。

对于是否给定对象应该按引用或按值传递的问题，应该根据它本身类的级别来决定。对于一些对象，按值传递无多大意义，对于其他一些对象却有很大的作用。通过正确设计类，我们可以适当处理这种情况。

首先，通过声明一个私有拷贝构造函数及一个私有赋值运算符，我们容易保护给定类的对象不按值传递。这样的方法并不是必需的。它们只是用于终止编译器在后台做一些我们不希望做的事。

假如我们想阻止 IStack 对象按值传递。我们要做的是在它的类定义中添加两行代码：第一行声明一个拷贝构造函数，第二行代码重载赋值运算符=。

```
class Istack
{
public:
  // ...
private:
    IStack(IStack const & src);
    IStack & operator =(IStack const & src);
  // ...
};
```

现在，尝试编译任何试图给某个函数传递 IStack、从某个函数返回一个 IStack 或者将一个 IStack 赋值给另外一个 IStack 的代码。它不会顺利编译!如果你在这种情况下忽略了 IStack 类名后的&，编译器就会立即发现此错误。

所以，你应该给创建的每个类添加拷贝构造函数和赋值重载吗？坦白地说，我认为这并不实际。的确有一些类被传递——它们应该在声明中具有这种类型的保护。对于其他一些类，则要应用自己的判断力具体分析。

不过，有一些按值传递的类的对象发挥了一定的作用。这些类具有值语义。通常，此类对象的“足迹”相当小，但并非总是如此。如果你关心的是操作性能，那么大小是一个重要因素。

而且，按值传递对象的替代办法通常是使用 new 不断从自由存储区分配新拷贝——这是一种开销更大的选择。所以，即使对一些较大的对象，你也不应该废弃值语义。例如，对于

矢量或矩阵这样一些代数类就是如此。

在多数情况下，给予一个类值语义并不需要做任何事。正如我们在前面看到的，编译器将自动提供一个缺省拷贝构造函数以及缺省赋值运算符。如果缺省浅层拷贝适宜于复制所有保存在对象中的相关数据，你就达到了目的。

一条有用的经验是，如果类有一个特殊的析构函数，你应该提供一个拷贝构造函数以及重载赋值运算符。

当浅层复制不足以完成任务时，你必须提供适当的拷贝构造函数并重载运算符。让我们考虑这样一个简单的例子：类 StringVal 的对象表示一个可以按值安全传递的字符串。

Value

```
class StringVal
{
  friend StringVal operator+(StringVal const &v1,
                             StringVal const &v2);
public:
  StringVal(char const * cstr)
    :_buf(0)
  {
    std::cout << "Constructor taking char * "
              << cstr << std::endl;
    Init(cstr);
  }
  StringVal(StringVal const & str)
    :_buf(0)
  {
    std::cout << "Copy constructor "
              << str.c_str()<< std::endl;
    Init(str.c_str());
  }
  ~StringVal()
  {
    std::cout << "Destructor of " << _buf << std::endl;
    delete _buf;
  }
  StringVal & operator=(StringVal const & str);
  char const * c_str()const { return _buf; }
private:
  StringVal(): _buf(0){}
  void Init(char const * cstr)
  {
    _buf = new char [std::strlen(cstr)+ 1];
    std::strcpy(_buf, cstr);
```

```
  }
private:
  char * _buf;
};
```

这里发生了很多事。首先，我们得到了许多构造函数。让我们利用此机会来学习不同类型的构造函数。

- 无参数的缺省构造函数。如果你想动态分配这个类的对象的数组，就必须包含这一构造函数。缺省构造函数也可以是带有缺省参数值的构造函数。对于 StringVal，缺省构造函数就是私有的，所以，只可以定义带有初始化器列表的静态数组。
- 带单个参数的构造函数。除非声明为显式，否则，此构造函数提供从它的参数类型到类所表示类型的隐式转换。在这里，我们让编译器将任何 char*转换成 StringVal（如果此转换对代码有意义的话）。稍后，我们会举一些这样的例子。单参数构造函数也可以用于对象定义的语义变种中，其中使用一个等号来给构造函数传递参数，如下所示：

```
Value val = "Foo";
```

- 拷贝构造函数。它带有一个指向同一个类中某对象的常量引用并克隆它。

接下来，我们重载了赋值运算符。根据约定，赋值运算符带有一个对某对象的常量引用并将它复制到同一个类的现有对象中。与拷贝构造函数不一样，赋值运算符必须能作为目标处理一个已初始化对象。所以，对此需要有特殊的防范。我们通常必须释放目标对象保存的任何资源，而且分配新资源来保存源的拷贝。

```
StringVal & StringVal::operator=(StringVal const & str)
{
  std::cout << "Operator = " << str.c_str()<< std::endl;
  if(this != &str)
  {
    if(std::strlen(_buf)< std::strlen(str.c_str()))
    {
      delete _buf;
      Init(str.c_str());
    }
    else
      std::strcpy(_buf, str.c_str());
  }
  return *this;
}
```

对于下面完全合法的赋值用法需要特殊的代码来处理：

```
Value val("Foo");
val = val;
```

在这里——赋值运算符的定义中，如果它不是为了测试“源等于目标”，我们可以在进行复制前释放此源。

还有一件事你必须知道：赋值运算符不是继承而来的——你必须为每个派生类提供单独

的重载赋值运算符。如果你认真加以思考，就会发现意义所在。使用基类 operatir =给派生类的对象赋值可能导致对象限制——只复制基类中的部分成员数据。如果一个派生类添加了新的数据成员，它们不会被复制。

最后，为了让事情变得有趣。我为 StringVal 类型的对象添加了一个加运算符重载。

```
StringVal operator +(StringVal const &v1,
                     StringVal const &v2)
{
  std::cout << " operator +(" << v1.c_str()<< ", "
    << v2.c_str()<< ")\n";

  StringVal result; // 空
  int len = std::strlen(v1._buf)+ std::strlen(v2._buf);
  char * buf = new char [len + 1];
  std::strcpy(buf, v1.c_str());
  std::strcat(buf, v2.c_str());
  result._buf = buf;
  std::cout << " Returning by value\n";
  return result;
}
```

operator+带有两个指向 StringVal 的常量引用类型参数，而且按值返回另一个 StringVal（它们的“和”）。

现在，编译器每次看到 str1 + str2 之类的表达式，它就调用方法 StringVal::operator+。事实上，下面的两条语句是等价的：

```
str3 = str1 + str2;
str3.operator=(operator+(str1, str2));
```

对于我们人类来说，第一条语句通常容易理解得多。

注意，我没有声明 operator+为 StringVal 的一个方法。相反，它是一个自由函数。你可以在任何特定的对象外部执行它。这样做有其充足的理由，我将在后面进行解释。但首先，让我们演示另一种途径。

```
StringVal StringVal::operator+(StringVal const &v)
{
  std::cout << " operator +(" << c_str()<< ", "
          << v.c_str()<< ")\n";
  StringVal result; // 空
  int len = std::strlen(_buf)+ std::strlen(v._buf);
  char * buf = new char [len + 1];
  std::strcpy(buf, c_str());
  std::strcat(buf, v.c_str());
  result._buf = buf;
  std::cout << " Returning by value\n";
  return result;
}
```

在这里，operator+定义为 StringVal 的一个成员函数，所以，它只需要一个参数。第一个加数隐式为 this 对象。使用此版本 operator+的语法与自由函数相同，而且等价的工作过程如下：

```
str3 = str1 + str2;
str3.operator=(str1.operator+(str2));
```

两者之间仅有一个细微不同之处——在“方法”中两个加数并不是同等对待。这一般不会产生问题，除非你试图相加一个 char*与一个 StringVal 对象时会发生问题。请看下面语句：

```
StringVal str("Foo");
StringVal vSum = str + "bar";
```

一般而言，你不需要定义一个 operator+的单独重载来处理此类混合加法。例如，如果使用自由函数来解决：

```
StringVal operator+(StringVal const & str1,
                    char const * str2);
```

或使用“方法”途径：

```
StringVal StringVal::operator+(char const * str);
```

你看到问题所在了吗？使用两种办法都可以相加一个 a char*与一个 StringVal。但是如果你想反过来相加，即在一个 char*上加一个 StringVal，也只有自由函数才能胜任：

```
StringVal operator+(char const * str1,
                    StringVal const & str2);
StringVal str("bar");
StringVal strSum = "Foo" + str;
```

顺便提醒你，请注意编译器如何灵活地解析算术表达式。它查看操作数的类型，然后根据此类型查找正确的重载运算符。实际上，它甚至比这更聪明!

我说过，你需要一个特殊的 operator+重载来处理混合加法。这不完全正确。事实上，即使没有任何其他 operator+重载，给 StringVal 上加 char*与在 char*上加 StringVal 都能正常执行。那如何实现呢？因为编译器可以执行从 char*到 StringVal 的隐式转换，编译器使用此转换来解析一个混合算术表达式。在一个双精度数上加上一个整数具有一定的概括性（int 转换成 double）。例如：

```
StringVal str1("Foo");
char * str2 = "bar"
StringVal result = str1 + str2;
```

编译器可以自由地将 str2 转换成一个 StringVal，而且使用重载 operator+，它带有两个指向 StringVal 的常量引用。如果你想了解从 char*到 StringVal 的隐式转换，请再次查看 StringVal 构造函数集。请记住：

> 带有一个参数的构造函数定义了一个从它的参数类型到它自身类型的隐式转换。

但是从一个 char*转换至 StringVal 是另外一回事。此时，编译器必须做更多的事。它必须将 char*转换成一个对 StringVal 的常量引用。编译器通过创建一个临时 StringVal，用一个

char*初始化而且将它的常量引用传递给被调用函数来完成。需要注意的是必须让 StringVal 在调用期间保持有效。编译器如何完成的具体过程暂时不作讲解，只要知道它能正常工作就行了。

如果你使用一个非常量引用来完成相同的任务，哪些没有发挥作用呢？带有一个非常量引用的函数期望能附带更改此参数的值。所以，使用一个临时量来过渡是不可能的。事实上，编译器拒绝对非常量引用执行隐式转换，甚至对内置的基本类型也是如此。请看下面的例子：

```
void increment(double & x){ x += 1.0; }
int n = 5;
increment(n); // <- 错误!
```

编译器无法知道你期望 n 的值从 5 变为 6。而且它不能传递 int 的地址（在此地址要求有一个 double），因为两者的内存布局完全不同（或者在大小上也不同）。编译器因此告诉你这样的信息“Error: There's no conversion from int to（non-const）reference to double”（错误：从 int 到 double 的（非常量）引用无转换路径）。

与之相比，下面的代码可被顺利编译：

```
void dontIncrement(double const & x);
int n = 5;
dontIncrement(n); // <- OK!
```

因为编译器可以自由生成一个临时 double，用 n 的值初始化并给它传递一个常量引用。

最后要说明的事：为什么我声明 operator=返回一个对 StringVal 的引用，而且 operator+按值返回 StringVal？让我们先看 operator=，它才必须有返回（按引用或按值）。

```
StringVal s1("Foo"), s2("Bar"), s3("Baz");
s1 = s2 = s3;
```

或者

```
if(s1 = s2)// 注意，并非 ==
  ...
```

赋值链能正常工作，因为它被解释成：

```
s1.operator=(s2.operator=(s3));
```

而且将赋值解释成条件来工作，因为它等价于：

```
if(s1.operator=(s2))
  ...
```

从 operator=返回的值被解释成 true（当它不为 0 时）。

另一方面，如果你不喜欢此类编程方式，你最好声明 operator=无返回。

```
Void String Val::operator=(String Val const & str)
```

当重载 operator+时我不会推荐此方式，因为我们通常对相加的结果感兴趣。相反，我想解释的是为什么优先选择按值返回相加的结果，而不采用按引用返回。原因很简单——请问自己“引用什么呢？”它不能引用任何加数，因为它们要保持其原始值。因此，你必须创建一个新对象来保存加法的结果。如果你作为本地变量在堆栈中创建它，它从 operator+返回后将很快消失。最终你将得到一个指向某些堆栈垃圾的引用。如果你使用 new 来创建它，它不

会在你的引用之下消失，但你将永远不能删除它。最终会导致内存泄漏。

将 operator+（或其他任何二元运算符）看作一个双参数构造函数。它构建了一个新的实体，两个加数的和，而且它必须将此实体放在某个地方。将此新值放在何处的选择余地很少，尤其是当你处理一个复杂的算术表达式时。按值返回它并让编译器负责如下工作。

小结：基本内置类型通常按值传递。使用一个拷贝构造函数及一个重载 operator=，完成一个针对用户自定义类型的同类值语义。除非明确覆盖，否则编译器将为用户自定义类型创建一个缺省拷贝构造函数以及缺省赋值。

第二部分

技　术

一个程序员改进现有代码所花的平均时间比编写新代码所花时间要多得多。这就是为什么程序修改技巧如此重要的原因。程序修改包括维护（排错）、添加新功能、重写以及重新构建。如果你不断添加新功能，你必须安排重新编写的时间，否则，你的程序将变得越来越难以维护。重写代码是程序员应付混乱的唯一法宝。

在本书的这一部分内容中，我将指导读者对符号计算器进行许多主要及次要代码的重写（此程序为本书第一部分所创建的程序）。我利用重写代码之机向读者描述一些重要的编程技术并介绍C++的一些更高级功能。

第6章 清 理

我们希望程序能尽量按设计来编写，但我们在实现具体细节时，总是会临时添加一些元素。我们创建辅助类与函数，还要给现有类添加方法和数据成员。这一过程回想起来就像发酵培养般地增长一样。随着每个单元的增大，它就开始“膨胀”和“萌芽”。最后，新芽分离并开始另一个生命。

在软件中，增大和膨胀（尤其如此）现象的发生相当自然。“萌芽”与“分离”则需要程序员付出一定的努力。代码复审对这一过程帮助极大。

在下面的内容中，我们将看到过于增大的文件、方法以及类。我们应该创建一个新文件、新方法，尤其是要创建一个新类。我们要时刻预备将来的代码重用。当复审代码时，我们还要找出哪一部分需要更全面的解释，并添加适当的注释。

Projct CalcTech

本书这一部分内容的源代码位于 Code Co-op 项目 CalcTech 中。从光盘的启动界面上选择 CalcTech 链接来创建此项目。边注标明了下一步要展开的代码。

6.1 输出解耦

Script 1

让我们再来看计算器的主循环。

```
std::cout << "> "; // 提示
std::cin.getline(buf, maxBuf);
Scanner scanner(buf);
Parser parser(scanner, store, funTab, symTab);
status = parser.Eval();
```

头两行负责用户输入，接下来的三行负责解析并求值。我们丢失了其他代码吗？难道计算器不应该显示计算结果？当然，应该显示——它在 Eval 内完成。将计算结果发送到输入是求值过程的一个附带作用。但一般情况下：

附带功能并不好!

为了理解计算器的所作所为，别人必须查阅 Eval 内的代码。

听起来不太理想，但为了知道它的作用，你难道不必查看 Eval 的内部实现代码吗？否则，你如何知道它对用户输入首先解析然后求值、用户输入通过扫描器传递给解析呢？

好了，事情况就是这样。实际上，我们为什么不用更透明的方式重写主循环中的最行一行呢？

```
std::cout << "> ";  // 提示
std::cin.getline(buf, maxBuf);
Scanner scanner(buf);
Parser  parser(scanner, store, funTab, symTab);
status = parser.Parse();
double result = parser.Calculate();
std::cout << result << std::endl;
```

这里，每条语句只完成一个定义明确的任务。我不必知道其他任何对象的实现细节就可以阅读此代码。我阅读此代码的方式为：

1. 提示用户。
2. 获利一行输入。
3. 创建封装一行输入的扫描器。
4. 创建一个解析器来处理此输入。
5. 让解析器解析输入。
6. 让解析器计算结果。
7. 显示结果。

此方案从美学上讲让人更乐意阅读，但仅仅用美学不足以说服别人改写他们的代码。学习将美学变成实际参数是一件好事。所以我认为此版本比原始版本更好。原因有以下几条：

- 它分离了三个定义明确的动作：解析、计算以及显示。解耦后的代码更易维护。
- 输入和输出在同一级执行，而不是像原来那样，输入在顶级执行但输出分布在低级代码中。
- 输入和输出为用户界面的一部分，它通常独立于程序的其他部分。将它们集中放在一个地方是一个好习惯，其原因就在此。我已经准备让程序从命令行界面转换到一个 Windows 界面了。

让 Parser 支持这一新的分解方案所做的修改相当明显。我们必须删除旧的 Parse 方法并将它直接粘贴到调用它的唯一地方（即代码_pTree = Expr()处）。然后，我们将 Eval 改名为

Parse 并将计算移到新方法 Calculate 中。我们还去掉了 Execute。

作为一个好的附带作用，我们还可以在错误检查中更正一个漏洞：输入行不应该在表达式后包含任何字符。以前，像 1 + 1 = 1 这样的表达式被顺利接受。现在，IsDone 方法将返回 false，并提示错误的发生。

```
Status Parser::Parse()
{
  // 一切都是表达式
  _pTree = Expr();
  if(!_scanner.IsDone())
    _status = stError;
  return _status;
}

double Parser::Calculate()const
{
  assert(_status == stOk);
  assert(_pTree != 0);
  return  _pTree->Calc();
}

bool Scanner::IsDone()const
{
  return _buf [_iLook] == '\0';
}
```

我们学生说服我，如果解析导致一个错误，就不应该调用 Calculate。正如你所见，这样做的确简化了 Parse 和 Calculate 的代码。现在，我们必须在 main 中检查错误，不过这是可行的。在那里有一段状态代码可供使用，那我们为什么不应用它来做一些有用的事呢？

而且只要我们利用它，还可以检查扫描器的空行而不影响此情况下解析器的创建。有时，我们还可以利用这一点把空行当成一个用户退出计算器的符号。一切都准备就绪了。

```
std::cout << "\nEnter empty line to quit\n";
// 一次处理一行输入
do
{
std::cout << "> ";  // 提示
std::cin.getline(buf, maxBuf); // 读一行
Scanner scanner(buf); // 创建扫描器
if(!scanner.IsEmpty())
{
  // 创建解析器
  Parser  parser(scanner, store, funTab, symTab);
  status = parser.Parse();
  if(status == stOk)
  {
```

```
      double result = parser.Calculate();
      std::cout << result << std::endl;
    }
    else
    {
      std::cout << "Syntax error.\n";
    }
  }
  else
  {
    break;
  }
} while(status != stQuit);
```

看看我们所做的修改：我们花很小的气力重新组织了程序的顶层。这可是传统编程的一大禁忌。修改程序顶层就是重写整个程序的同义词。如果我们想要成功，必须做一些适当的工作。当然，这是一个小程序，而且除此之外，此程序由我写成，所以，我就容易对它进行修改。我可以保证，我在项目中已经试验此方法不少于百次了，它完全能正常工作。我甚至于让一个新手在此项目中修改顶层并扩展到最底层作相应修改。坦白地说，如果我没有这样做，我将永远不会发现一些设计上的缺陷——就像我们刚才在计算器程序中找到的错误一样。

Script 2

整个程序中还有其他几个输出操作。它们大部分用于错误报告。因为我们对错误报告和错误传播尚无一个清楚的计划，所以我们只能做一个美化性的修改。我们不将错误信息直接发送到标准输出 cout，而是将它们直接发送到标准错误，如下所示：

```
std::cerr << "Error: division by zero\n";
```

唯一的实际不同之处是：如果你重定向程序的标准输出到一个文件，错误信息将仍然在控制台显示。这不是我们关心的所在，问题是，当我们使程序 Windows 化后，我们要能迅速检索 cerr 对应的源，例如，将错误输出重定向到消息框。

6.2 与“防错性编程”做斗争

“防错性编程（Defensive programming）”听起来不错，真是如此吗？防错性程序员编写的程序应该更安全、更可靠，这正确吗？

不！防错性编程是一种危险的工程恶习（这是自我贬低的表现，我将很快在我的代码中列举一些这样的例子）。防错性编程提倡编写支持正常的代码，甚至于在程序员出错时也如此。实际上，此编程法允许 bug 有机会覆盖它们的踪迹。通宵捕捉 bug 的程序员，如果他不是防错性编程，也许他早就找出 bug 了，也不会因为防错性编程而作出无谓牺牲了。在大部分情况下，防错性编程就是不能准确理解程序不同部分间契约的结果。一个糊涂或懒惰的程序员，如果不尽量去理解程序的逻辑，他就可能写出什么情况下都能工作的程

序。此代码会抹杀程序的逻辑，甚至使程序的进一步开发更混乱。程序员最终无法把握整个代码。

防错性编程经常遇到的一个混淆根源是字符串的长度与适当数组大小间的矛盾。数组必须有足够的空间存放终止 null 符，而此终止 null 并没有计入字符串的长度内。系统头中定义的 MAX_PATH 考虑此终止 null 符了吗？当分配一个缓冲器时，程序员应该使用 MAX_PATH，还是应该使用 MAX_PATH+1 呢？



Script 3

现在审查计算器程序中的这种混淆。Scanner::GetSymbolName 假定它的参数 len 为能适应缓冲器的字符串的最大值。所以，缓冲器大小应该比 len 大 1。因此有下面的断言：

```
assert(len >= maxSymLen);
assert(_lenSymbol <= maxSymLen);
```

但是，当我们从解析器调用 GetSymbolName 时，我们分配的是一个 maxSymLen 字符的缓冲器，并作为它的长度传递 maxSymLen。这看起来像一个 bug，不是吗？我们刚刚说过，GetSymbolName 希望缓冲器比 len 长一个字符。

那么，它是如何工作的呢？事实表明，在扫描器中，我们防错性地删减了符号名字的长度，使其比 maxSymLen 少一个字符。

```
if(_lenSymbol >= maxSymLen)
  _lenSymbol = maxSymLen - 1;
```

所以现在的情况是，maxSymLen 包含终止 null，而 Scanner::GetSymbolName 希望 len 为缓冲器的大小，而不是字符串的长度。不过当它返回时，GetSymbolName 设置 len 为字符串的实际大小——不包含 null。嘿，它能工作！是的，它能工作，但这样的代码便于维护吗？我认为难以维护。

让我们合理化我们的假设。首先，maxSymLen 应该为不包含终止 null 的符号名的最大长度。用 null 终止是一个实现细节——我们应该能计数字符串而不必更改 maxSymLen 的值。因此方法 Scanner::Accept 应该按下面的方式删减符号名字：

```
if(_lenSymbol > maxSymLen)
  _lenSymbol = maxSymLen;
```

下面是 GetSymbolName 的调用方式：

```
char strSymbol [maxSymLen + 1];
int lenSym = _scanner.GetSymbolName(strSymbol,
                                    maxSymLen + 1);
```

下面是实现方式：

```
int Scanner::GetSymbolName(char * strOut, int lenBuf)
{
  assert(lenBuf > maxSymLen);
  assert(_lenSymbol < lenBuf);
  std::strncpy(strOut, &_buf[_iSymbol], _lenSymbol);
```

```
    strOut [_lenSymbol] = '\0';
    return _lenSymbol;
  }
```

内参数 lenBuf 为缓冲器的长度。返回值为复制到缓冲器中字符串的长度。这样就不太容易混淆了。

各种失败编程的另一个理由是出现了不应该发生的情况。计算器有一个内在的局限。例如，Store 的大小限制为 maxSymbols，设置值为 40。这已足够了，我们并不希望在正常情况下会超出此限制。即便如此，我们还是在 Store 中编写了一行防错性代码：

```
void Store::SetValue(int id, double val)
{
  if(id < _size)
  {
    _cell [id] = val;
    _status [id] = stInit;
  }
}
```

非常幸运，因为它让我们避免了解析器中的一些多余错误检查（稍后我们会看到）。

Script 4

当你降低对防错性编程的抵触后，你看到发生了什么？代码变得臃肿了。这种臃肿迟早会让你头疼。所以，让我们将此方法改成一种主动性编程——我们对（或其他人）代码的正确性如此有把握，因此我们可以毫不犹豫地运用断言。

```
void Store::SetValue(int id, double val)

{
  assert(id < _size);
  _cell [id] = val;
  _status [id] = stInit;
}
```

稍后我们将更正调用者的代码。

6.3　完备性编程的一个实例

Script 5

函数表的初始化是一个完备防错性编程的实例。首先，我们定义 maxIdFun 为函数项数组的大小。然后，我们定义数组大小 maxIdFun，但是我们用较少的项来初始化它。我们还用一个空指针及一个空字符串添加了一个信号。接着，在 FunctionTable 的构造函数中，我们设置循环从 0 开始一直到 maxIdFun – 1，但当我们碰到信号时跳出此循环。所有这一切用于避免函数数组大小和函数表中条目数量的不匹配。不过，使用下面的简单断言就足够了：

```
assert(maxIdFun ==
       sizeof FunctionArray/sizeof FunctionArray [0]);
```

sizeof运算符返回它的参数的大小（按字节）。当应用于一个静态数组时，它返回整个数组的大小。当应用于指针时，即使指针指向一个数组，它也返回指针的大小——在32位计算机上为4字节（sizeof的参数也是一个类型名，它将返回此类型对象的大小）。

请注意，为了让断言工作，我必须更正一个不便使用的名字冲突。FunctionTable构造函数的参数名与全局数组同名——它们都叫funArr。猜一下这将会怎样？当我首先编写此断言时，它并不按原来的设想工作：sizeof funArr一直返回4。我检查了引用若干次，尝试使用括号——依然如旧。我开始要怀疑编译器了，但我突然间意识到sizeof的作用对象是一个传递给构造函数的参数，而不是我所认为的全局数组。本地名funArr混淆了全局名funArr，因此，我处理的是指针而不是静态数组。

我得到了两个教训（确切地说是从n次失败中获得的教训）：不要对不同的变量使用同一个名字，而且如果运算符sizeof应用于数组时返回4，请不要怀疑编译器。这种讽刺性情况的发生往往是我按习惯（但不是这一次）以一个大写字母开始书写一个全局名字导致的（同时还使它不够简短）。

将全局数组名改成FunctionArray后，我将maxIdFun正确设置成14，而且让编译器通过计数初始化器调整数组的大小（我让方括号为空）。

```
const int maxIdFun = 14;
FunctionEntry FunctionArray [] =
{
  log, "log",
  ...
  atan, "atan"
};
```

现在，我们终于可以添加断言，它将捕捉FunctionArray大小与FunctionTable::_pFun指向数组之间的不匹配。

```
FunctionTable::FunctionTable(SymbolTable & symTab,
                             FunctionEntry * funArr)
  : _size(0)
{
  assert(maxIdFun ==
         sizeof FunctionArray / sizeof FunctionArray [0]);
  ...
}
```

当我给学生演示此方案时，他们立即注意到其中的问题：如果FunctionTable的构造函数通过一个指向与FunctionArray不同的数组的指针来调用的话，情况又将如何呢？没有办法阻止这种情况的发生。显然，设计工作从FunctionTable的一般性开始进化，原来的FunctionTable作为一个可以通过任何函数项数组初始化的对象，现在的发展方向是使FunctionArray成为FunctionTable的一个非常亲密的伙伴。事实上，这正是我在下面的代码改写中计划要做的。

下面是我为完成此目的编写的代码：

```
static FunctionEntry FunctionArray [] = { ... };
FunctionTable::FunctionTable(SymbolTable & symTab)
   : _size(0)
{
   assert(maxIdFun ==
         sizeof FunctionArray/sizeof FunctionArray [0]);
  ...
}
```

我让 FunctionArray 为静态（static），这意味着 FunctionArray 在定义它的文件 funtab.cpp 外部不可见。从这层意义上讲，静态（static）和外部（extern）表示不同的可见性。我所遵循的原则是：

> 数据隐藏几乎总是一个好主意。

顺便提及，现代 C++不鼓励针对数据隐藏使用静态。它的用途已经由名字空间（namespace）所代替（我们将在第 7 章讨论名字空间）。

能不能让 FunctionArray 和 FunctionTable 关系更密切？让 FunctionArray 成为类 FunctionTable 的一个成员如何？

现在要记住，FunctionArray 的整个思想是能静态地初始化——在编译时而不是在运行时初始化。C++中有一种对应结构称做静态成员（注意此关键字的重载）。

> 一个静态成员可以看作一个类对象的成员而不是一个特定实例对象的成员。在面向对象的编程语言中（如 Smalltalk），在实例对象和类对象间有着明显的区别。同一个类可以有许多对象的实例，但它们都共享同一个类对象。一个类对象可能有自己的数据成员和方法。类对象数据成员的值在所有实例间共享。不必访问任何实例对象就可以访问类数据或调用类方法。类方法仅可以操作于类数据，而不能操作于特定实例的数据之上。
>
> 虽然 C++没有类对象记号，但静态数据成员及静态方法能有效提供等价功能。每个类的每个静态数据成员只有一个实例，而不管给定类创建了多少个对象（事实上，即使没有创建实例，静态数据成员仍然可访问）。一个静态方法实质上是一个常规函数（它没有 this 指针），它通过在类中的成员关系限制了可见性。

声明一个私有静态数据成员的语法为：

```
class FunctionTable
{
  ...
private:
  static FunctionEntry  _functionArray [];
  ...
};
```

而且它在实现文件中的初始化方式为：

```
FunctionEntry FunctionTable::_functionArray [] =
{
  log, "log",
  ...
};
```

请注意，即使_functionArray 为私有，它的初始化也是在全局作用域内完成的。

不幸的是，当_functionArray 通过一个未定义大小（通过声明中的空方括号可知）声明为静态成员时，sizeof 运算符不会返回像我们希望的那样返回初始化的数组大小。编译器只是查看 FunctionTable 中的声明，而且它不会在那里查找与大小有关的信息。经过仔细考虑，你才会明白它的实际意义。

还有另一个有趣的建议。为什么不使用 sizeof FunctionArray/sizeof FunctionArray [0]作为 FunctionTable 声明中数组_pFun 的大小？

```
Pfun _pFun [sizeof FunctionArray / sizeof FunctionArray[0]];
```

从原理上讲这是可能的，但实际上我们必须在声明 FunctionTable 之前初始化 FunctionArray，因而编译器可以计算 sizeof。这将意味着将初始化放在头文件中，而且因为头必须包含在若干实现文件中，所以会发生一个连接器错误（多重定义）。

这一提案被修改成让 FunctionTable 成为一个常量数组，于是连接器不会认为它在多个文件中初始化了。毕竟，对于如下结构：

```
const int maxLen = 10;
```

在头文件中是完全可以接受的，如

```
const FunctionEntry FunctionArray [] =
{
  log, "log",
    ...
};
```

此方案不错，因为它使得函数数组增长时函数指针表自动调整大小。这比利用断言且必须人工调整大小要好。不过，仅在内部初始化函数表时需要的 FunctionArray 对整个程序突然变成可见的了。这一点与其他方案截然不同，其他方案让此数组成为函数表的一个静态成员。它违背了数据隐藏的原则。

Script 6

那么，我们该如何做呢？答案是：让我们在实现文件 funtab.cpp 中隐藏函数数组。但不是不惜一切代价来静态调整_pFun 大小，而是使用动态分配。在分配之时，编译器就知道 FunctionTable 的静态大小了。

```
FunctionTable::FunctionTable(SymbolTable& symTab)
  : _size(sizeof FunctionArray / sizeof FunctionArray [0])
{
  _pFun = new PtrFun [_size];
```

```
for(int i = 0; i < _size; ++i)
{
  int len =  std::strlen(FunctionArray[i].strFun);
  _pFun [i] = FunctionArray [i].pFun;
  std::cerr << FunctionArray[i].strFun << std::endl;
  int j = symTab.ForceAdd(FunctionArray[i].strFun, len);
  assert(i == j);
}
```

不要忘记在析构函数中释放此数组：

```
FunctionTable::~FunctionTable()
{
  delete []_pFun;
}
```

我们已经见过了几个类形成模式（堆栈及其序列生成器）的例子。下面我们将看到一个静态数组与一个类协作的例子。实际上，这种模式将共享相同的头和实现文件。

选择此方案后我失去了什么？我失去了数组的静态分配。动态分配的缺陷是双重的：必须记住释放对象——这是维护上的缺陷，还有访问代价上的极小增加，我在这里提及这一点仅仅是为了讲解的完整性，因为这一点实在可以忽略不计。反过来，我可以隐藏函数数组及其初始化器列表，这再次有利于重要的可维护性。

FunctionTable 的代码不可以立即重用，但通过更改构造函数让它接受函数数组项及项的计数容易达到此目的。因为我觉得它没有多大的重用性，所以没有这样做。如此一来，我将减少最高抽象层（main 级别）可见对象的数量。

6.4 补充

边界情况与错误路径的处理是初级程序员常犯错误的地方。不检查数组边界、不测试错误返回、忽略空指针等等错误类型都不会在肤浅的测试中表现出来。除了非常特殊的情况外，程序同样能正常工作。当开发专业软件时，此方式就不可接受了。特殊情况与主要执行路径同等重要——事实上，它们都需要大量的代码来处理。

Script 7

我在解析器中发现了一个草率错误检查的例子。如果没有足够的空间存放符号变量，方法 SymbolTable::ForceAdd 可能会失败。然而我们没有在解析器中测试此现象。代码能工作（有几分）是因为在符号表中应用了防错性编程。既然我们已经用进攻性代码代替了防错性代码，我们也必须更正此错误：

```
// Factor := Ident
if(id == idNotFound)
{
```

```
// 在符号表中添加新标识符
id = _symTab.ForceAdd(strSymbol, lenSym);
if(id == idNotFound)
{
  std::cerr << "Error: Too many variables\n";
  _status = stError;
  pNode = 0;
}
if(id != idNotFound)
   pNode = new VarNode(id, _store);
```

一般经验是尽可能立即检查此错误。在这里，我们不必等到 VarNode 创建后再去破坏这一个类的契约（VarNode::Calc 的执行动作并不正确）。

6.5 改进类之间的通信

当给一个函数传递字符串时，同时传递它的长度有时有利——尤其是由于其他一些原因已经知晓长度的时候。另一方面，我们可能在函数内部计算此字符串的长度。这是以速度换取可维护性。如果长度已经知道，为什么调用 strlen 呢？通过传递值，我们引入了另一个可能犯的错误——传递错误的长度。



Script 8

让我们看看方法 SymbolTable::ForceAdd。它从几个地方被调用，连同一个字符串与其长度。我们开始要防止这样的代码：

```
int id = symTab.ForceAdd("e", 1);
...
id = symTab.ForceAdd("pi", 2);
```

从操作性能的立场看，它的意义不大，而且犯错误的风险高，尤其因为这部分代码将来要进行扩展。

于是，在 Function::Table 的构造函数中，我们编写了计算 strlen 且只将它传递给 ForceAdd 的代码。

```
int len =  std::strlen(FunctionArray[i].strFun);
...
int j = symTab.ForceAdd(FunctionArray[i].strFun, len);
```

最后，对于 Parser::Factor，我们可能认为有充足的理由使用此结构，在这里，作为调用 Scanner::GetSymbolName 的一个附带作用让字符串的长度可用。但是这种节省是可笑的——即使只增加了极少引入 bug 的可能性也的确不值得。

所以，有必要修改 ForceAdd 的接口与实现代码。

```
int SymbolTable::ForceAdd(char const * str)

{
  int len = std::strlen(str);
  assert(len > 0);
  ...
}
```

结果，我们不需要从 Scanner:: GetSymbolName 返回符号的大小。

Script 9

下面让我们进一步探讨符号表中不同类间的交互作用。哈希表是一个列表数组，而且在哈希表中检索字符串返回一个短列表。符号表代码重复遍历此短列表，而且试图使用直接比较找到此字符串。下面是完成此任务的代码：

```
int SymbolTable::Find(char const * str)const
{
  // 从哈希表获得短列表
  List const & list = _htab.Find(str);
  //遍历列表
  for(Link const * pLink = list.GetHead();
      pLink != 0;
      pLink = pLink->Next())
  {
    int id = pLink->Id();
    int offStr = _offStr [id];
    if(_strBuf.IsEqual(offStr, str))
      return id;
  }
  return idNotFound;
}
```

类 List 的对象担任着哈希表与符号表之间通信包的角色。作为一个代码审查人员，我为这一点感到不安。如此安排会有一些不利的影响——HTable 必须暴露它的实现细节（如链表）给客户？毕竟，可能有哈希表没有使用链表。作为代码的审查者，即使我们不知道如果改善这一状况，我也可以表达我的担忧。不过，在这里，我知道如何来做。所有哈希表的一个共有特征就是它们必须能处理冲突（除非它们是完美的哈希表——我们就不必担心这些事）。因此，它们总是（至少从概念上讲）返回一个溢出列表。客户必须要做的是按顺序通过此列表以完成检索。

这里的关键字是序列。一个针对链表的序列生成器对象容易实现。它只是我们在上面所见的代码的重新包装。但是序列生成器还有更多的作用。我们已经见过堆栈序列生成器。程序员也可以为一个数组编写一个序列生成器，一个跳过条目的序列生成器等。我们可以让哈希表返回一个序列生成器而不是一个列表。这样一来，我们将自己与另一个哈希表实现细节分离（如果你想知道为什么我叫它序列生成器，而不叫迭代器，那是因为我将迭代器这一名

字保留给 C++标准库中与指针类似的对象——我们将在适当的时间学习它)。

图 6-1 显示了客户代码(SymbolTable 代码)与哈希表的交互作用，哈希表使用序列生成器对象封装哈希表搜索结果。

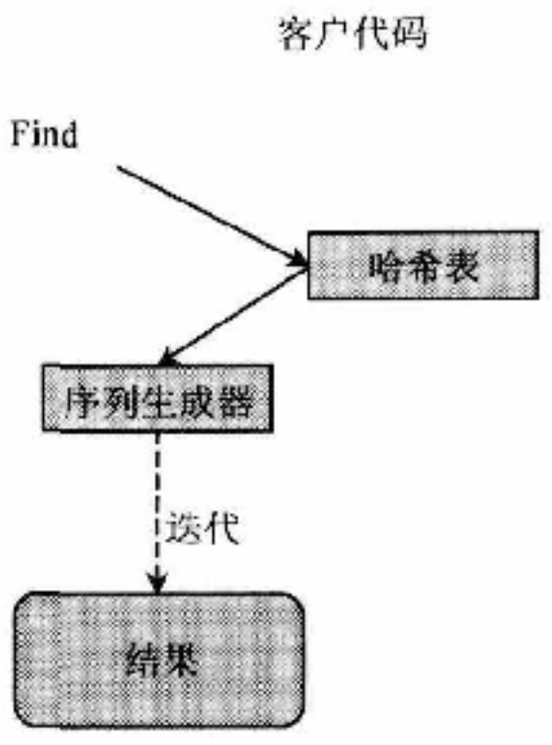


图 6-1 在哈希表和客户间使用序列生成器通信

简单翻译图 6-1 可能得到如下代码:

```
ListSeq * HashTable::Find(char const * str);// BAD!!!
```

由于某些原因，序列生成器在 Find 中使用 new 来分配，而且在排序完成后立即删除。

千万不要这样做！我们的代码简单可靠，但它变成了一个危险的组装系统。不要笑！在一些商业库或 API 集中，你可能会找到这种代码实例。

我们的解决方案是对象倒置。不是在被调用者的内部创建辅助对象并让调用者负责删除它，我们将让调用者创建一个辅助本地对象(在堆栈上)，在退出此作用域时自动释放此对象。此对象的构造函数将带有一个对源对象的引用(在这里，源对象就是哈希表)，而且通过它进行数据的传输。事实上，当我们首先介绍序列生成器时，此机制已经在发挥作用(这是一个堆栈序列生成器，它的构造函数带有一个指向堆栈对象的引用)。

下面是 SymbolTable::Find 使用这样一个序列生成器 IdSeq 后的新实现代码。它带有一个指向哈希表的引用，并且从它获得了一个对应于字符串 str 的列表。

```
int SymbolTable::Find(char const * str)const
{
  // 从哈希表遍历短列表
  for(IdSeq seq(_htab, str);
      !seq.AtEnd();
      seq.Advance())
  {
    int id = seq.GetId();
    int offStr = _offStr [id];
    if(_strBuf.IsEqual(offStr, str))
      return id;
  }
  return idNotFound;
}
```

这就是从客户的角度观察到的。通过哈希表进行检索简化为创建一个对应于哈希表、字符串对的序列生成器。换句话说，我们从一种情况(发送到哈希表的字符串生成一个序列生成器)变换到了另一种等价情况(序列生成器创建于哈希表与字符串间的交互作用)。

我仍然没有解释整个实现过程。显然，IdSeq 是一个特殊的类，它知道哈希表的一些工作方式。它在暗中还从哈希表获得一个链表，而且将它用于生成序列。区别在于，哈希表不再对任何人公布此链表。HTable::Find 方法为私有而且仅能被友元 IdSeq 访问(IdSeq 现在是

Htable 的友元)。

```
class Htable
{
  friend class IdSeq;
  ...
private:
  List & Find(char const * str)const;
  ...
};
```

还有一个不太重要的问题：链表序列生成器是一个真正的更一般性的类，也就是说，它应归于文件 list.h 而且与哈希表无关。一般这样的元素才完美：

```
// 列表序列生成器
// 用法:
// for(ListSeq seq(list);
//      !seq.AtEnd();
//      seq.Advance())
// {
//    int id = seq.GetId();
//    ...
// }
class ListSeq
{
public:
  ListSeq(List const & list)
   :  _pLink(list.GetHead()){}
  bool AtEnd()const { return _pLink == 0; }
  void Advance(){ _pLink = _pLink->Next(); }
  int GetId()const { return _pLink->Id(); }
private:
  Link const * _pLink; // Current link
};
```

不过，在我们的特殊例子中，我们需要一个特殊化的序列生成器在它的构造函数中与哈希表交互作用。

没有比这更简单的了。我们可以轻松创建一个从一般性序列生成器继承了所有特性的序列生成器。它只添加自己的构造函数，它从哈希表获取链表并将它传递给父代的构造函数。

```
// 短列表序列生成器
// 客户创建序列生成器
// 检索给定字符
class IdSeq: public ListSeq
{
public:
   IdSeq(HTable const & htab, char const * str)
    : ListSeq(htab.Find(str)){}
};
```

下面总结这一情况。在这一出演出中只有 5 个演员，所以它仍然可以将它们都记住。表 6-1 列出了所有这些演员，并解释了它们的角色。

表 6-1　　哈希表相关类及其关系

类	文　件	角　　色
List	list.h	单链表
ListSeq	list.h	单链表通用序列生成器
IdSeq	htab.h	专业化序列生成器：HTable 的友元，从 ListSeq 继承行为
HTable	htab.h	哈希表。对于客户只能通过 IdSeq 访问。对于友元，私有方法 Find 返回一个 List
SymbolTable	symtab.h	HTable 的客户。只能通过专业化的 IdSeq 访问 HTable

请注意类是如何分组的：List 和 ListSeq 作为一个组。它们为一般性类，所以可以用于以后的重用。IdSeq 和 HTable 形成了另外一个组，它的实现依赖于 List 组。它们也可用于重用。SymbolTable 依靠 HTable 和几个对象来保存字符串。

完成这些修改后，如何评价我们的成就？假设我们决定将哈希表的实现改成所谓的闭合哈希表。按这种方案，溢出项与直接项保存在同一个数组中。溢出列表不再是一个链表，而是一条如何跳转此数组的指令。它可以封装在一个序列生成器中（当然，它必须为哈希表的友元）。

那么，将符号表从一个哈希表的实现转换成另一个哈希表的元程序是什么？假想我们让新类的名字与前面相同（HTable 和 IdSeq），元程序为空。我们不必修改符号表中的一行代码。这真是一个成就！

6.6 更正设计缺陷

Script 10

正如我以前提到的，算术表达式的解析器有一个缺陷——它的语法为右结合。这就意味着像下面这样的表达式：

```
8 - 2 + 1
```

将被解析成：

```
8 -(2 + 1)= 5
```

而不是我们所期望的：

```
(8 - 2)+ 1 = 7
```

我们可以尝试更正这一语法，让它包含下面类型的左结合性形式：

```
Expression := Expression '+' Term
```

但是它们将立即将递归解析器送到一个无限循环中（Expr 通过调用 Expr 来启动，Expr 又通过调用 Expr 启动……）。有一些更强大的解析算法可以处理左递归，但它们对于我们这一简单的计算器来说太复杂了。不过，也有一个折衷的方案将结合性从语法转变成执行。根据此途径，表达式解析器不停地逐项解析，并将它们都保存在一个多节点新类型中。多节点后代的所有项都具有相同的状态，但当它用于计算时，则是从左到右进行（左结合）。例如，解析下面的表达式：

```
8 - 2 + 1
```

将得到图 6-2 所示的树。

多节点 SumNode 的方法 Calc 从左到右求和，得到正确答案：

```
8 - 2 + 1 = 7
```

我们可以在解析项时运用同样的技巧来更正因子的结合性。

将语法修改为包含如下类型形式（这称做扩展 Backus-Naur Form（BNF）标记）。

```
Expr  := Term {('+' | '-')Term }
Term := Factor {('*' | '/')Factor }
```

或者更准确地描述为，表达式是一个后面有 0 个或多个符号（加号或减号）及一个项组合的项。反过来，项是一个后面有 0 个或多个乘号或除号及一个因子组合的因子。喜欢用根据图示进行描述的读者可以参考图 6-3，它用示意图表示了这种关系。

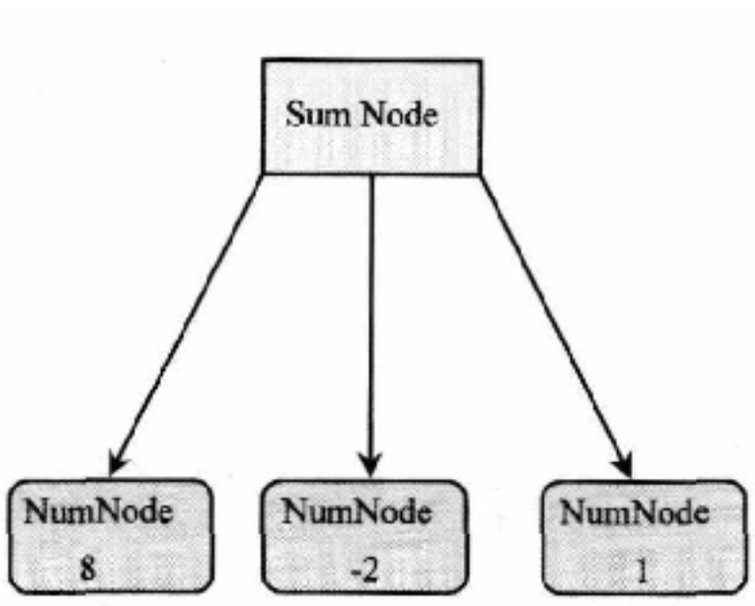


图 6-2 使用多节点解析 8-2+1 的结果

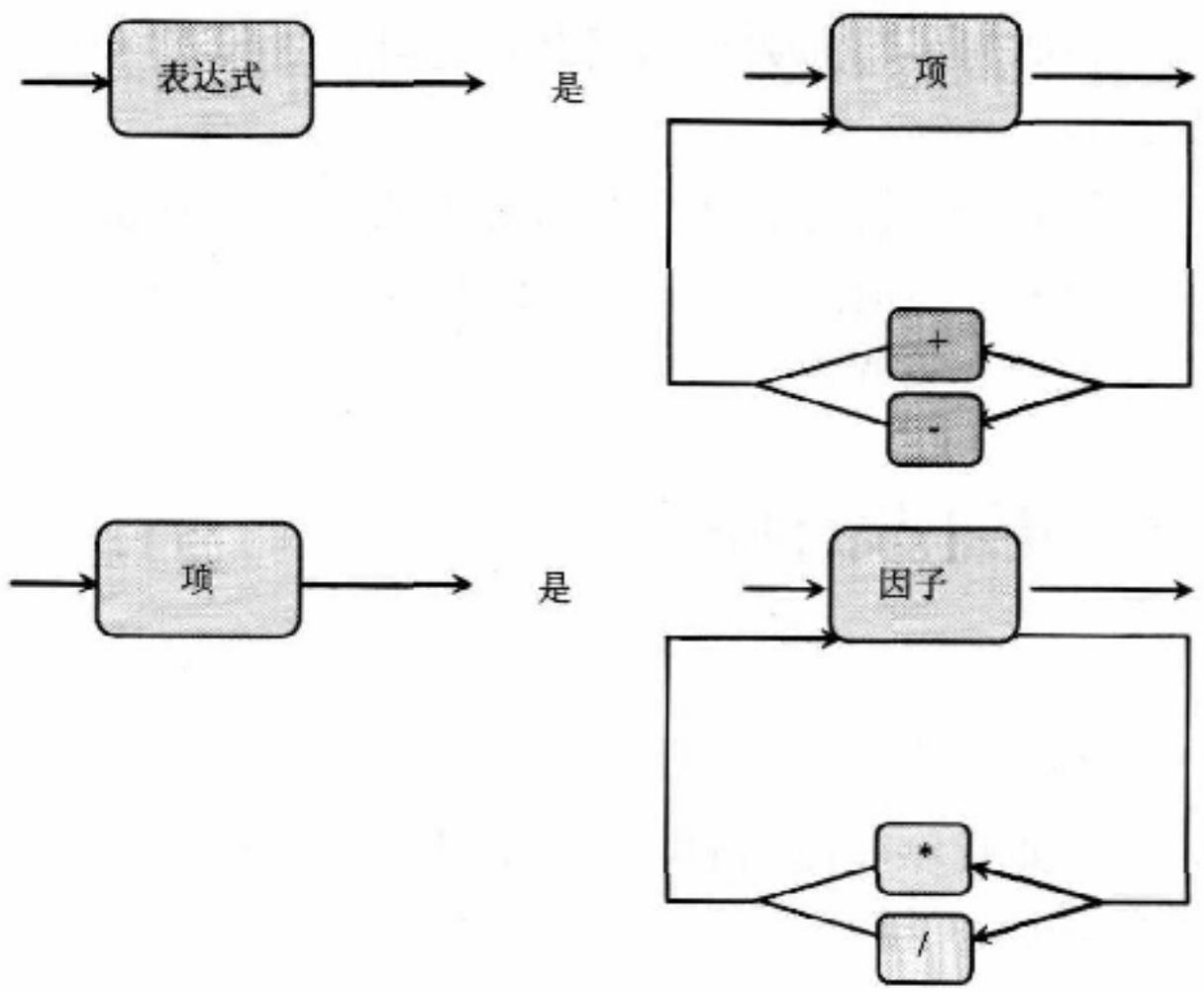


图 6-3 表达式和项的扩展形式

“0 个或多个……”的表述对应于以上样式中的大括号，直接将它翻译成递归解析器中的 do/while 循环。下面是 Parser::Expr 的正确实现代码：

```
Node * Parser::Expr()
{
```

```
Node * pNode = Term();
EToken token = _scanner.Token();
if(token == tPlus || token == tMinus)
{
  // Expr := Term {('+' | '-')Term }
  MultiNode * pMultiNode = new SumNode(pNode);
  do
  {
    _scanner.Accept();
    Node * pRight = Term();
    pMultiNode->AddChild(pRight,(token == tPlus));
    token = _scanner.Token();
  } while(token == tPlus || token == tMinus);
  pNode = pMultiNode;
}
else if(token == tAssign)
{
  _scanner.Accept();
  // Expr := Term = Expr
  Node * pRight = Expr();
  // 若此项为一个左值
  if(pNode->IsLvalue())
  {
    pNode = new AssignNode(pNode, pRight);
  }
  else
  {
    _status = stError;
    delete pNode;
    pNode = Expr();
  }
 }
 return pNode;
}
```

Parser::Term 的编码经历了相似的转换。

```
Node * Parser::Term()
{
  Node * pNode = Factor();
  EToken token = _scanner.Token();
  if(token == tMult || token == tDivide)
  {
    // Term := Factor {('*' | '/')Factor }
    MultiNode * pMultiNode = new ProductNode(pNode);
    do
    {
      _scanner.Accept();
```

```
      Node * pRight = Factor();
      pMultiNode->AddChild(pRight,(token == tMult));
      token = _scanner.Token();
    } while(token == tMult || token == tDivide);
    pNode = pMultiNode;
  }
  return pNode;
}
```

我们已经引入了两个新 Node 类型：SumNode 和 ProductNode。它们都共享可以有多子代这一属性，而且，这些子代有一个“符号”与它们相关联。在 SumNode 中，此符号为加号或减号；在 ProductNode 中，此符号为乘号或除号。我们将在某个数组中保存指向子代的指针，并将对应的符号保存在另一个数组中。约定是：在_isPositive 布尔数组中，将加号和乘号保存为 true，而减号和除号保存为 false。在（抽象）基类 MultiNode 的派生类中，这一区别很明显。MultiNode 表示 SumNode 和 ProductNode 间的共性。对于我们的第一个不太精确的实现代码，我们将随意把子代数量限制为 8 个——这一不足之处将在第 9 章更正。精确与否，我们可以测试当用户编写一个包含 8 个以上子节点的表达式时是否溢出并返回一个错误。

```
const int MAX_CHILDREN = 8;
// 一般性多节点: 一个抽象类
class MultiNode: public Node
{
public:
  MultiNode(Node * pNode)
    : _isError(false)
  {
    _aChild [0] = pNode;
    _aIsPositive[0] = true;
    _iCur = 1;
  }
  ~MultiNode();
  void AddChild(Node * pNode, bool isPositive)
  {
    if(_iCur == MAX_CHILDREN)
    {
      _isError = true;
      return;
    }
    _aChild [_iCur] = pNode;
    _aIsPositive [_iCur] = isPositive;
    ++_iCur;
  }
protected:
  bool   _isError;
  int    _iCur;
```

```
  Node *  _aChild [MAX_CHILDREN];
  bool    _aIsPositive [MAX_CHILDREN];
};
```

一个布尔值的数组可以按一个 bool 类型的数组得以最简单实现。不过，它不是最节省空间的实现办法。为了保存两个可能值（truc 和 false），所需的只是一个比特。标准 C++库在它自己的布尔数组实现代码中应用了这一优化。

下面是两个派生类的定义。请注意，因为 MultiNode 为一个 Node，所以也是 SumNode 和 ProductNode（is-a 关系的传递）。

```
// 求和
class SumNode: public MultiNode
{
public:
  SumNode(Node * pNode)
    : MultiNode(pNode){}
  double Calc()const;
};

// 乘和除
// 这里的符号指指数：正表示乘，负表示除
class ProductNode: public MultiNode
{
public:
  ProductNode(Node * pNode)
    : MultiNode(pNode){}
  double Calc()const;
};
```

完成后，不要忘记删除子代。

```
MultiNode::~MultiNode()
{
  for(int i = 0; i < _iCur; ++i)
     delete _aChild [i];
}
```

下面该讲我们的主题了：使代数具有左结合性的 Calc 方法。

```
double SumNode::Calc()const
{
  if(_isError)
  {
    std::cerr << "Error: too many terms\n";
    return 0.0;
  }

  double sum = 0.0;
  for(int i = 0; i < _iCur; ++i)
```

```
  {
    double val = _aChild [i]->Calc();
    if(_aIsPositive [i])
     sum += val;
    else
     sum -= val;
  }
  return sum;
}
//注意: 计算为左结合性
double ProductNode::Calc()const
{
  if(_isError)
  {
    std::cerr << "Error: too many terms\n";
    return 0.0;
  }

  double prod = 1.0;
  for(int i = 0; i < _iCur; ++i)
  {
    double val = _aChild [i]->Calc();
    if(_aIsPositive [i])
      prod *= val;
    else if(val != 0.0)
      prod /= val;
    else
    {
      std::cerr << "Error: division by zero\n";
      return HUGE_VAL;
    }
  }
  return prod;
}
```

注意这一修改也同时解决了如下表达式的问题：

```
1 + x = 2
```

它们不再合法，因为一个表达式的解析是“贪婪的”。它首先吞掉等号左边的所有项，然后将它们放到一个节点中。不过，此节点不是一个左值，故而不会创建赋值节点，并且出现一个语法错误。

script 11

此时，你应该打开 script 11，此代码是我们讲解“清理”的一个总结。

第7章

隐藏实现细节

嵌入类、受保护构造函数、隐藏常量、匿名枚举、名字空间。

一个好的软件工程师就像一个间谍。他或她根据“是否有必要知道”的原则向同事报告消息，因为同事知道太多会让他们惹上麻烦。我不是在这里警告工业间谍的危险——我要说的是与复杂性的不懈斗争。隐藏的细节越多，就越容易理解它的作用。

7.1 使用嵌入类

Script 12

类 Link 仅由链表以及它的友元序列生成器在内部使用。坦白地说，甚至没人知道它的存在。在类 List 外部重用这一个类的可能性极小。我们将在 List 类定义的私有部分隐藏 Link 的定义。

```
class List
{
  friend class ListSeq;
public:
  List();
  ~List();
  void Add(int id);
private:
  // 嵌套类定义
  class Link
  {
  public:
    Link(Link * pNext, int id)
        : _pNext(pNext), _id(id){}
```

```
    Link *   Next()const { return _pNext;}
    int       Id()const { return _id;}
    private:
    int       _id;
    Link    * _pNext;
  };
private:
  Link const * GetHead()const { return _pHead; }
  Link*_pHead;
};
```

类嵌入的语法具有自解释性。请注意，一个嵌入类不能访问嵌入类的数据成员或方法。

类 ListSeq 有一个数据成员，它是一个指向 Link 的指针。作为 List 的友元，它可以顺利访问类 Link 的私有定义。不过，在 Link 名字前必须加上类 List 的名字——新名字就是 List::Link。

```
class ListSeq
{
public:
  bool AtEnd()const { return _pLink == 0; }
  void Advance(){ _pLink = _pLink->Next(); }
  int GetId()const { return _pLink->Id(); }
protected:
  ListSeq(List const & list)
    : _pLink(list.GetHead()){}
private:
  List::Link const *_pLink;
};
```

注意在类 List 的定义内，我们不必添加名字 Link。只是在定义的外部，我们才必须区分 List 的内部类 List::Link 与可能出现的一些调用 Link 的外部类。事实上，如果我们想在 List 内部访问一个外部类 Link，我们必须在它的前面放置一个双冒号（::Link）来与外部 Link 明确区分开。

一种查看类声明的办法是将它看作一个方法、数据成员以及类型的集合。到目前为止，我们只处理过方法及数据成员；现在我们可以学习如何在一个类中定义另一个类型——在此例中，就是另一个类。一个类也可以使用 typedef 为其他类型创建别名。我们将后面内容中看到更多应用了此技术的例子。

类 List 及 ListSeq 经过了一些额外的私有化（对于 ListSeq 来说，它大概应该称做“受保护化”）。我声明 GetHead 方法为私有，但是让 ListSeq 为一个友元，因而，它仍然可以调用 GetHead。我还声明 ListSeq 的构造函数为受保护，因为我们从未在程序中创建它——我们只使用派生类 IdSeq 的对象。

我可能在这里使用了过多的私有化，使得这些更难以重用。不过，重要的是，在作出适

可而止的明智选择之前，你应该知道你可以在程序中使用私有化的程度。

7.2 组合类

Script 13

序列生成器对象与列表对象从概念上联系非常紧密。从某种程度上讲，ListSeq 是 List 的一个友元就是代码中这种关系的一种反映。但是，我们还可以做得更好——可以在列表类中嵌入序列生成器类。不过，这一次我们不想让它成为私有——我们希望 List 的所有客户都能使用它。你应该知道，在嵌入类的外部，客户只能通过添加外部类的名字作为前缀来访问嵌入类。在本例中，作用域辨别前缀是 List::。因而将嵌入类的名字缩短为 Seq 有一定的意义。在外部，它以 List::Seq 的样子出现，而在 List 的内部，再也没有名字冲突的危险。

下面是 List 修改后的声明：

```
class List
{
public:
  List();
  ~List();
  void Add(int id);
private:
  class Link
  {
  public:
    Link(Link * pNext, int id)
      : _pNext(pNext), _id(id){}

    Link *  Next()const { return _pNext; }
    int     Id()const { return _id; }
  private:
    Link *  _pNext;
    int     _id;
  };
public:
  class Seq
  {
  public:
    bool AtEnd()const { return _pLink == 0; }
    void Advance(){ _pLink = _pLink->Next(); }
    int GetId()const { return _pLink->Id(); }

  protected:
```

```
    Seq(List const & list)
      : _pLink(list.GetHead()){}
  private:
  };
    Link const * _pLink;      // 当前链接
  friend class Seq;
private:
  Link const * GetHead()const { return _pHead; }
    Link * _pHead;
};
```

顺便提醒一句，请注意标签 public、private 和 protected 可以按任意顺序混合使用。

即使类 Seq 嵌入类 List 中，它也没有任何访问特权。特别是，如果它不声明为 friend 的话，它仍然不能调用 List 的私有 GetHead。

序列生成器的唯一客户为哈希表序列生成器 IdSeq。因而我们必须修改它的定义。

```
class IdSeq: public List::Seq
{
public:
  IdSeq(HTable const & htab, char const * str)
     : List::Seq(htab.Find(str)){}
};
```

而且，在这种情况下，如果我们将它所属类的定义移到类 HTable 的内部，将会如何呢？我们甚至可以将名字从 IdSeq 缩短为 Seq，而且使它以 HTable::Seq 的身份对外部分可见。如从前一样，序列生成器必须为它的外部类 HTable 的友元。如果我们在嵌入类 Seq 的定义前声明此友元关系，我们就必须使用如下名字：

```
friend class HTable::Seq;
```

此定义后的同样声明可简化为：

```
friend class Seq;
```

在 SymbolTable::Find reflects 内的用法反映了新的命名结构：

```
for(HTable::Seq seq(_htab, str);
       !seq.AtEnd();
    seq.Advance())
```

7.3 使用名字空间组合

程序中还有多个需要更紧密组合的关联实体集。但这一次，我们讨论的是类和数据的混合。我所指的是 FunctionTable、FunctionEntry 以及 FunctionArray 的联合体（我们还可以添加 CoTan 的定义，它在函数表之外的场合从未使用）。当然，我可以在 FunctionTable 内部嵌入 FunctionEntry，使 CoTan 成为一个静态方法，而且声明 FunctionArray 为一个静态成员（我们在前面内容中已讨论过这一话题）。



Script 14

不过，还有一个更好的解决方案。在 C++中，我们可以创建一个更高级别的组别，它称做名字空间（namespace）。看看我们将要组合的对象名字。除了 CoTan 外，其他所有名字都共有一个共同的前缀：Function。所以，让我们命名此名字空间为 Function 并在其中嵌入类 Table 的定义（前面已讲过的 FunctionTable）。

```
namespace Function
{
  class Table
  {
  public:
    Table(SymbolTable & symTab);
    ~Table(){ delete []_pFun; }
    int Size()const { return _size; }
    PFun GetFun(int id){ return _pFun [id]; }
  private:
    PFun * _pFun;
    int    _size;
  };
}
```

名字空间的优点在于你可以在实现文件中延续使用它。下面是文件 funtab.cpp 的精减版本：

```
namespace Function
{
  double CoTan(double x){...}
  class Entry {...};
  Entry Array [] =
  {...};
  Table::Table(SymbolTable & symTab)
    : _size(sizeof Array / sizeof Array [0])
  {...}
}
```

你已经猜到了，下一步是在程序的其余部分用 Function::Table 替换所有的 FunctionTable。唯一的需要注意的地方是头 parse.h 中的前向声明。你不能只使用 class Function::Table;，因为编译器没有看到 Function 名字空间的声明（记住，使用前向声明是为了避免包含 funtab.h）。我们必须告诉编译器不仅 Table 是一个类，而且它还在 Function 名字空间中进行了声明。完成这一任务的代码是：

```
do it:
namespace Function
{
  class Table;
}
```

```
class Parser
{
public:
  Parser(Scanner & scanner,
         Store & store,
         Function::Table & funTab,
         SymbolTable & symTab);
  ...
};
```

一般原则是，许多类、函数及对象的传统命名约定，在现代 C++中被作用域分辨前缀（scope resolution prefix）代替，而且它与嵌入类和名字空间的使用相关联。

顺便提到的是，整个 C++标准库都由名字空间包装，它的名字为 std。现在，你应该最终明白 cin、cout、endl 等前缀 std::的含义了。这是一种避免前缀使用关键字的途径。我不推荐这一种做法，但你可以在文件的顶部使用如下声明：

```
using namespace std;
```

而且放弃使用 std::作用域分辨前缀。你也可以有更多的选择，声明使用某特殊类型、函数或对象，如

```
using std::cout;
```

在本书的后面章节中，我将介绍更多名字空间的使用。

7.4　在枚举中隐藏常量

针对特定类的实现，程序中应用了几种相应的常量。使用它们可以在类的定义中自然地隐藏这些常量的定义。我们可以使用枚举来完成。我们甚至不必给枚举命名——它们可以为匿名。

了解一下在 C++中有多少种定义常量的途径。有旧式的 C #define 预处理程序宏，有类型安全全局 const，最后还有最小作用域 enum。哪一种最好？这完全取决于类型。如果你需要一个指定类型的常量，也就是说，需要一个 double 或一个（用户自定义）矢量，那就使用全局 const。如果你只需要一个般性的整数类型常量——如对于数组边界——请看看它的作用域。如果只是一个类需要，或一组紧密相关类需要，就应该使用一个 enum。如果它的作用域较大，则应使用一个全局 const int。另一方面，#define 可用于越过类型检查或类型转换——无论如何都要避免这种情况的发生（我们将在讨论宏时再讲解这一点）。

Script 15

下面是使用枚举隐藏常量的第一个例子。常量 idNotFound 对应于 SymbolTable：

```
class SymbolTable
{
public:
```

```
    //嵌入匿名枚举
    enum { idNotFound = -1 };
    ...
}
```

在实现 Find 时无需作任何修改，因为它是 SymbolTable 的一个方法，Find 可以不受任何限制地访问 idNotFound。

对于解析器并不如此。它仍然可以使用常量 idNotFound，因为它在 SymbolTable 的公有部分定义，但它的名字前面必须添加它所嵌入类的名字。

```
if(_scanner.Token()== tLParen)          // 函数调用
{
  _scanner.Accept();                    // 接受 '('
  pNode = Expr();
  if(_scanner.Token()== tRParen)
    _scanner.Accept();                  // 接受 ')'
  else
    _status = stError;
  // 使用嵌入枚举
  if(id != SymbolTable::idNotFound
      && id < _funTab.Size())
  {
    pNode = new FunNode(
      _funTab.GetFun(id), pNode);
  }
  else
  {
    std::cerr << "Unknown function \"";
    std::cerr << strSymbol << "\"\n";
  }
}
else
{
  // Factor := Ident
  if(id == SymbolTable::idNotFound)
  {
    // 给符号表添加新的标识符
    id = _symTab.ForceAdd(strSymbol);
    if(id == SymbolTable::idNotFound)
    {
      std::cerr << "Error: Too many variables\n";
      _status = stError;
      pNode = 0;
    }
  }
  if(id != SymbolTable::idNotFound)
    pNode = new VarNode(id, _store);
}
```

最大符号长度可看作 Scanner 的内部限制（虽然有些人可能认为它是它所识别“语言”的限制——我们将在后面消除这一限制）。

```
class Scanner
{
public:
  // 嵌入匿名枚举
  enum { maxSymLen = 80 };
  ...
};
```

7.5　在本地变量中隐藏常量

Script 16

一般情况下，仅使用于某一个代码片断中的常量不应该暴露在整个全局作用域中，它应该在应用它的作用域内定义。编译器仍然会内联这样的常量（也就是说，它将在这一常量名出现的地方用其字面值替换，而不是在内存中引入一个单独的变量）。有几个常量仅用于 main 中：

```
int main()
{
  const int maxBuf = 100;
  const int maxSymbols = 40;
  char buf [maxBuf];
  Status status;
  SymbolTable symTab(maxSymbols);
}
```

Script 17

这是对我们第一次尝试隐藏代码的总结。

第8章 共　享

你注意到我们在代码改写中交替使用全局与本地对象了吗？无论何时谈到共享，多数情况下我们是指全局。只要谈到数据或实现隐藏，则是指本地。下面让我们讨论共享这一主题。

8.1 分离全局程序参数

Script 18

从全局的角度看，我们在本地作用域中经常看到的一些常量实际上是程序的可调参数。假设有人（或者说客户）想增加符号名字的最大长度，或者希望能给符号表添加更多的项，或者非常需要增加输入缓冲器的大小。我们不得不深入程序中的类和函数中去查找对应的常量。

幸运的是，有几种途径可以让这种精细的调整变得简单一些。我只介绍最基本的一种办法——用一个文件 Params.h 搜集所有可调常量。下面是这一文件的内容：

```
const int maxBuf = 100;          // 输入缓冲器的大小
const int maxSymbols = 40;       // 符号表的大小
const int maxSymLen = 80;        // 符号名字的最大长度
```

程序中几乎总是存在一些可调参数，它们的改变需要重新编译。它们位于 Params.h 之中。

通常还有另外一套由用户调整的参数。它们称做用户参数选择，这些参数永久保存在一些便利的位置。即使这些参数有自己的缺省值，也同样放在 Params.h 中。

8.2 增强包容性

我们知道，我们的程序有一些内在限制：看看文件 Params.h 就会明白。显然，我们希望这些限制越少越好。我们为符号、缓冲器中的字符等分配足够的空间，使用户在日常操作

中不要越过这些限制。不幸的是，这同时也意味着：程序员在测试中不可能触及这些极限。而且确信无疑的是——即使我们没有触及任何边界，但用户可能会！

让我们做一个小试验。编辑文件 Params.h 并将所有的极限设置成一些相当低的值。我们甚至可以使用两套参数——一套用于常规操作，另一套用于测试。我使用条件编译指令在两套参数间切换。在#if 指令中，将 0 变成 1 就可以切换回常规参数。

```
#if 0
const int maxBuf = 100;       // 输入缓冲器的大小
const int maxSymbols = 40;    // 符号表的大小
const int maxSymLen = 80;     // 符号名字的最大长度
#else
const int maxBuf = 8;         // 输入缓冲器的大小
const int maxSymbols = 5;     //符号表的大小
const int maxSymLen = 4;      //符号名字的最大长度
#endif
```

重新编译整个程序后，运行它时立即碰到了 Function::Table 构造函数中的一个断言。检查后发现，程序试图将 14 个对应内置函数的符号装到符号表中，而符号表的最大容量只设置为 5。原来如此！显然，符号表的大小值为 5 太小了，但我们尚不能立即捕获这一错误。

一条断言足以防止这类问题的发生吗？一般来说，是可以的。只是当有人更改了 maxSymLen 的值或添加了更多的内置函数时才会发生这种特殊的问题。在这种情况下是安全的，因为

> 每次代码修改，程序都将在调试版本中重新生成（把断言打开），而且在发布给其他人之前至少运行一次。

这里所说的“其他人”并不是指客户——而是同一开发团队中的其他程序员。在发布程序给客户前，则需要更严格的测试。

因此，让我们将符号表的大小设置为 14——刚好可以容纳内置函数，而且尝试输入一个引入了一个新符号变量的表达式，例如 x = 0。天哪！程序崩溃了！

发生了什么？Parser::Factor 准确找出了问题所在，甚至还打印出信息“Error: Too many variables”（错误：太多变量），因而返回一个空指针。那它应该做什么呢？Parser::Factor 调用者并不希望得到一个空指针。让我们重新尝试吧！

为 Parser::Factor 所有调用者增加空指针检查（同时也针对 Parser::Expr 的调用者，它们可以传播从 Parser::Factor 和 Parser::Term 获得的 null）。

再次测试——这一次，一切工作正常；直到退出程序也没有发生任何问题。记住，我们一直都在调试器环境中运行程序的调试生成版本。一个好的调试运行时在程序释放内存时会进行堆检查。堆检查可以发现缓冲器溢出、重复删除等问题。实际上，的确如此！ Store 的构造函数发现了一个覆写内存块。通过“侦察”，终于发现了 bug 的蛛丝马迹。请看下面 Store 构造函数的代码：

```
int id = symTab.ForceAdd("e");
SetValue(id, std::exp(1));
std::cerr << "pi = " << 2 * std::acos(0.0)<< std::endl;
id = symTab.ForceAdd("pi");
SetValue(id, 2 * std::acos(0.0));
```

当符号表溢出时，它返回-1。Store::SetValue 通过-1 调用。

```
void SetValue(int id, double val)
{
  assert(id < _size);
  _cell [id] = val;
  _status [id] = stInit;
}
```

我足以断定 id 小于表的大小，但不能防止 id 的负值。

那好，将它改成下面代码：

```
assert(id >= 0 && id < _size);
```

再次编译运行。这一次，我们立即触及了断言。那就再进行一次 maxSymbols = 16 的测试，情况将更明朗。

现在，让我们测试输入缓冲器大小的限制。让我们输入一个大表达式，即一个有 10 个 0 的表达式。程序能工作，但与用户的期望值不一样。它首先打印不正确的结果，然后进入一个无限循环（在有些编译器上它能退出）。

首先，它为什么会进入一个无限循环？因为在第一次尝试时没有从输入读取所有的字符，所以下一次我们调用 cin.getline 时，它在没有填充缓冲器的情况下立即返回。一些库实现代码清除缓冲器，因而程序退出；而另外一些没有清除缓冲器，所以，它将永远循环下去。

不过，这不是最大的问题。程序不知不觉地返回错误值比崩溃更糟糕，不正确的信息比根本没有信息更坏。我们必须立即更正这一问题。首先，让我介绍一个特别常规的解决方案，它不必深究输入流的细节。

```
char buf [maxBuf + 1];
// ...
std::cin.getline(buf, maxBuf + 1);
if(std::strlen(buf)== maxBuf)
{
  std::cerr << "Error: Input buffer overflow\n";
  status = stError;
  break;
}
```

实现的办法就是将缓冲器的大小增加 1，于是我们可以获取比自身设置的极限多一个字符。现在，我们可以判断我们的要求是否得到了满足，如果得到了满足，就知道输入大于 maxBuf，问题就得以解决了。

在特殊情况下，可采用更简单的方案。当输入行不适应于缓冲器，对 getline 的调用不能满足时，流 cin 的状态就会改变。我们可以测试这一状态作出适当的反应。

```
std::cin.getline(buf, maxBuf); // 读取输入行
if(std::cin.fail())
{
  std::cerr << "Error: Input buffer overflow\n";
  status = stError;
  break;
}
```

现在进行最后的测试——最大符号长度。在测试之前，我们必须增加 maxSymbols 和 maxBuf，让我们可以给程序输入几个测试符号。我们发现，虽然程序能工作，但它却不知不觉欺骗了用户。尝试输入表达式 toolong = 6，然后显示变量 tool 的值。程编译器不会反对未初始化变量的使用，它将暗中将两个名字看作表示同一个变量。至少，我们可以在 Scanner::Accept 中显示一条警告。

```
if(_lenSymbol > maxSymLen)
{
   std::cerr << "Warning: Variable name truncated\n";
   _lenSymbol = maxSymLen;
}
```

可怕吧，不是吗？为什么程序的漏洞会如此之多？这是正常现象，或者只是一个例外？事实上，不管你多优秀，你开发的程序总会包含错误。编译器能找出其中的大部分简单错误——输入错误、类型不匹配、错误参数数量等。许多错误经过一些基本测试后显而易见。其余的错误就是“顽固的 bug”。

根除这些“顽固的 bug”有许多技术。一种技术是测试；特别是通过辅助代码可以使测试更容易，这些辅助代码的唯一目的就是捕获“bug”。我们在前面使用文件 Params.h 就是这样的目的。我将在后面内容中介绍更多的测试辅助技术。

这也与编写程序的态度有关。我编写此程序时没有太关心错误检查及异常条件。通常情况下我还是比较小心的，但这里，我是为了演示说明由于粗心导致的错误。

最后，还存在策略方面的因素。处理异常条件并非突发奇想——你必须制定一些规则。在后面，我们还要对这一问题作更多的探讨。

8.3 模板

Script 19

我们将类 List 与它的序列生成器放在一个单独的文件中，以便于将来的重用。在将来的任何时候，当我们需要一个整数链表时，我们可以包含这一测试合格的可靠代码。我刚才指的是“整数的链表”，如果我们需要的是一个无符号长整数或双精度数，甚至 Star 类型的链表呢？真正有用的是通过一种类型参数化的类。使用模板吧！

为了将 List 变成一个模板，你需要在类定义之前添加：

```
template<class T>
```

或者使用等价的：

```
template<typename T>
```

并用 T 代替 int。我们用参数 T 参数化 List 的定义，T 代表任何类型。一旦定义了模板，你可以对它进行实例化——用任何类型代替 T 后来使用它，它甚至不必是一个类。例如：

```
List<int> myList;
```

将定义一个名叫 myList 的整数 List。你也可以按自己的需要随意声明其他列表，如无符号长整数的列表 List<unsigned long>、22 精度数列表 List<double>，或者 Star 的列表 List<Star>。

下面是 List 模板的定义：

```
template<class T>
class List
{
public:
  List();
  ~List();
  void Add(T value);
private:
  class Link
  {
  public:
    Link(Link * pNext, T value)
      : _pNext(pNext), _value(value){}
    Link *  Next()const { return _pNext; }
    T GetValue()const { return _value; }
private:
    T       _value;
    Link *  _pNext;
  };
public:
  class Seq
  {
  public:
    bool AtEnd()const { return _pLink == 0; }
    void Advance(){ _pLink = _pLink->Next(); }
    T GetValue()const { return _pLink->GetValue(); }
protected:
    Seq(List const & list)
      : _pLink(list.GetHead()){}
private:
    Link const * _pLink; //  当前链接
  };
  friend class Seq;
private:
```

```
  Link const * GetHead()const { return _pHead; }
  Link * _pHead;
};
```

为了更一般化，我将读取保存值的方法名从 GetId 改为 GetValue。

由于技术方面的原因，现在的编译器要求所有的模板代码在模板实例化的位置均可见（这一要求可能在将来的编译器中有所变化）。这一要求的意思是，我们必须将所有的方法实现代码与模板类定义放在同一个头文件中。它没有让这些方法内联（除非它们被声明如此），也没有暗示它们的代码将被多次实例化。

下面是在类定义外部，但仍然在头文件 List.h 的内部定义构造函数的方式。请注意，只在双冒号前面的第一个 List 后面加上<T>。

```
template<class T>
List<T>: :List()
  : _pHead(0)
{}
```

析构函数同样如此：

```
template <class T>
List <T>::~List()
{
  // 释放链表
  while(_pHead != 0)
  {
    Link * pLink = _pHead;
    _pHead = _pHead->Next();
    delete pLink;
  }
}
```

其他方法遵循同样的模式：

```
template <class T>
void List <T>::Add(T value)
{
  Link * pLink = new Link(_pHead, value);
  _pHead = pLink;
}
```

“模板化”的第二部分是用 List<int>替换所有的整数 List。例如下面的哈希表：

```
class Htable
{
public:
  class Seq: public List<int>::Seq
  {
  public:
    Seq(HTable const & htab, char const * str)
      : List<int>::Seq(htab.Find(str)){}
  };
```

```
  friend class Seq;
public:
  explicit HTable(int size): _size(size)
  {
    _aList = new List<int> [size];
  }
  ~HTable()
  {
    delete []_aList;
  }

  void Add(char const * str, int id);
private:
  List<int> const & Find(char const * str)const;
  int hash(char const * str)const;

  List<int> *_aList;
  int        _size;
};
```

定义一个模板本身不会在可执行程序中生成任何代码。模板就像针对编译器的一套指令：在需要的时候，它告诉编译器如何生成代码。编译器第一次看到给定模板的实例化时，它就创建所需的代码。看到在 HTable 内使用 List<int>将提示编译器生成类 List 的定义，它已针对整数型作了适当的修改。将生成非内联方法（列表的构造函数、析构函数以及 Add）的代码。但是，要是我们也实例化 List<double>，则会针对 double 生成一套完全独立的构造函数、析构函数及 Add 版本。

一般而言，一个类模板可以通过多个类型参数化。它也可以通过值参数化。例如：

```
template<class T, int size>
class Array
{
public:
  T _arr [size];
};
```

注意，在这里，通过不同参数值的两个实例化

```
Array<double, 10> a;
Array<double, 20> b;
```

等价于两个独立的类定义。如果这些类有与之相关联的代码（非内联方法），这些代码也要被复制。

除了类模板外，C++还提供了函数模板——通过类型或值参数化函数。max 的定义就是一个典型的函数模板例子：

```
template<class T>
inline T const & max(T const & a, T const & b)
{
```

```
  if(a < b)
    return b;
  else
    return a;
}
```

要计算两个整数的最大值，我们就可以使用如下代码：

```
int i = max<int>(j, 5);
```

或者完全忽略类型并让编译器基于参数类型判断应该使用哪个版本的 max 来实例化：

```
int i = max(j, 5);
```

请注意：因为函数 max 定义为内联，使用模板来代替宏就没有隐藏的开销（我们将在后面讨论宏）。



Script 20

小结：一个类模板是通过一个或多个类型参数化的类定义。类参数在关键字 template 后面的尖括号中列出。

```
template <class type1, class type2 ...>
```

为了实例化模板，用实际的类型（内置或用户自定义均可）替换类型参数就行了。你还可以定义函数模板并且通过值参数化模板。

练习

1. 将简单堆栈的实现过程（请参见 1.8 节的“抽象数据类型”）转换成通过值类型及堆栈的最大容量参数化的模板：

```
template<class T, int maxSize>
class Stack
```

2. 将动态堆栈（请参见 3.5.1“动态堆栈”一节内容）转换成一个模板。

3. 我已经定义了一个整数哈希表。将此定义转换成一个通过保存在其中的值类型参数化的模板。针对 double 实例化此模板。

第9章 去掉限制

动态数组、数组访问运算符重载。

当你试图保存几个小时的编辑结果，却只看到消息框提示“不能保存：打开了太多的窗口”或其他一些类似信息时，难道你不恼怒吗？不幸的是，这种情形常常发生——自认为是程序员的人硬性用一些定长数组来保存重要的程序资源。模板的出现使除去没有必要的限制变得如此容易，而这些限制是几乎没有任何办法去除的。

让我们首先查看没有应用模板时，我们在程序中哪些地方使用了不合理的限制条件。我在 MultiNode 中使用了这样一个限制，在这里，我随意将子代的数量限制为 8 个以下。整个符号表充满了这种限制，例如，字符串的缓冲器受到限制，而且偏移的数组也有限。最后，变量值的保存也存在一个随意指定的极限值。

你可能会问“用户触及这些限制点的可能性有多少？为了处理这些不可能的事例，值得我们花气力来编写模板吗？”答案是：

使用动态容器比通过设置限制来解决问题麻烦会更少。

请想想你必须显示的那些错误信息、这些错误的传播（顺便提一下，我将在后面章节中单独讨论这些主题）以及信息字符串的定位。但实际上，你根本不想陷于这种泥潭中！

9.1 动态数组

Script 21

几乎所有人工设置限制的情形都可通过一个模板来解决——这就是动态数组。让我们学习动态数组的设计方法及实现方式。在这一过程中，你不仅要学习更多的模板和运算符重载知识，还能为学习 C++标准库中的一般目的动态 vector 作准备。

让我们首先对这一数据结构提出一系列的要求。我们希望动态数组的查找与工作方式尽量与数组相同。这意味着我们可以通过索引来访问它的元素，包括读与写。添加新元素有些技巧性。由于操作性能的原因，我们不希望每次访问数组时都检查其边界（除非针对调试版本），而且如果需要可以扩展此数组。同样，在本程序中，实际上我们从未需要在某任意偏移处添加新元素——我们总是在数组的结尾处添加元素。它对跟踪最后添加的元素有利，并补充方法 Push 并扩展此数组。

在这一特殊的练习中，我决定让它更加一般化一点儿，而且添加方法 Set，让它在一个任意偏移处添加新元素。我这样做的原因是为了能使用共享相同插入索引的成对同步数组——稍后，我将举例说明。

还有一件事：在许多情况下，有必要用一些缺省值来预填充数组。例如，一个指针数组可以用空指针预填充，整数可以预设为 0 或-1、一些缺省值的枚举等。我们将在动态数组中保存这一缺省值，所以，当数组增长时，我们可以使用它来预填充数组的新块。

```
template <class T>
class DynArray
{
  enum { initDynArray = 8 };
public:
  explicit DynArray(T const valDefault);
  ~DynArray();
  void Set(int i, T const val);
  T operator [](int i)const;
  T & operator [](int i);
  bool InRange(int i)const { return i < _capacity; }
private:
  void Grow(int i);

  T     * _arr;
  int     _capacity; // 数组大小
  T const _valDefault;
};
```

代码行

```
T operator [](int i)const;
```

是另一个运算符重载的例子。我们在这里重载了数组访问 operator []（方括号），目的是像常规数组一样访问动态数组。

注意：DynArray 类型的对象不具有值语义（内置数组类型也没有——你不能按值传递整个数组）。它们不应该被复制或用来赋值。由编译器完成的缺省浅层复制将不能正常发挥作用——它将导致双重删除。如果你在意不小心复制的 DynArray 对象，你可以提供一个拷贝构造函数的哑元未实现声明以及一个赋值运算符（我们已经在 5.13“值语义”一节中讨论过这部分内容）：

```
DynArray(DynArray const & src);
DynArray & operator =(DynArray const & src);
```

下面是定义一个整数动态数组、使用 Add 给它添加值以及使用索引读取值的方式：

```
DynArray<int> a(0); // 缺省值为 0 的动态数组
Set(3, 10);  // 在偏移 3 处添加值为 10 的一项
int i = a [3];  // 像常规数组一样访问
```

编译器将最后一行翻译成对 operator []的调用，如下所示：

```
int i = a.operator [](3);
```

这是另一种正确的语法。

注意，我们在同一个类中还重载了同一个运算符：

```
T & operator [](int i);
```

如何实现这一点呢？同一个运算符，带有相同的参数，但它通过引用返回 T，而不是通过值。当编译器看到 a[3]时，它能够在两者之间正确选择吗？答案是不能——不允许只通过返回类型来区分重载。真正能明确区分的是关键字 const。一个方法为常量，而另一个不是。当作用于一个常量对象时，编译器将选择常量方法，否则，它将使用非常量方法。

我们可以通过非常量、按引用返回的运算符版本将数组索引的结果用作一个左值。下面的语句是合法的：

```
a [3] = 11;
```

对应更冗长的版本是：

```
int & r = a.operator [](3); //引用第 3 个元素
r = 11; // 通过引用更改值
```

在上面的例子中，如果 a 定义为一个 const，编译器将把它标记为一个错误。而且这也的确不对！你不应该更改一个常量对象的值。

以上说明了为什么我要定义两个版本的方法。如果我只定义非常量版本，我就不能应用于常量对象或 DynArray 的常量方法。事实上，我们的程序正是为了此原因而不能编译。

如果我只定义常量版本，我就不能使用索引修改数组的元素——我不能将它们用作左值。

还有第三种可能性：你可能会说，通过返回一个引用，即使它是一个非常量引用，你也不会真正地更改对象的状态。

```
T & operator [](int i)const { return _arr[i]; }
```

C++遵循这一严格的常量定义，它的作用是双方面的。正面作用是，你可以对常量与非常量对象只有一个 operator []重载。负面作用是，你很容易更改常量对象的内容。

动态数组的实现相当简单明了。下面是构造函数与析构函数。

```
template <class T>
DynArray <T>::DynArray(T const valDefault)
  : _capacity(initDynArray), _valDefault(valDefault)
```

```
{
  _arr = new T [initDynArray]; // 分配内存
  for(int i = 0; i < initDynArray; ++i)
    _arr[i] = _valDefault;
}

template <class T>
DynArray <T>::~DynArray()
{
  delete []_arr;
}
```

方法 Set 具有扩展数组的潜力。

```
template <class T>
void DynArray <T>::Set(int i, T const val)
{
  if(i >= _capacity)
    Grow(i);
  _arr [i] = val;
}
```

下面是两个版本的数组访问运算符。请注意，它们都不能用于扩展数组——断言将阻止这种行为。我们希望数组访问非常迅速（至少在非调试版本中如此），而每次访问时进行溢出测试、尝试扩展数组将强加大量开销。当我们希望在循环中逐个访问数组元素时，这一点非常重要。

```
template <class T>
inline T DynArray<T>::operator [](int i)const
{
  assert(i < _capacity);
  return _arr[i];
}
template <class T>
inline T & DynArray<T>::operator [](int i)
{
  assert(i < _capacity);
  return _arr[i];
}
```

Grow 方法的目的是让数组大小加倍。不过，当我们请求在加倍之上再扩展数组时，它也会不得不这样做。

```
//私有非内联方法
template <class T>
void DynArray <T>::Grow(int idxMax)
{
  int newSize = 2 * _capacity;
  if(idxMax >= newSize)
    newSize = idxMax + 1;
```

```
// 分配新数组
T * arrNew = new T [newSize];
// 复制所有项
int i;
for(i = 0; i < _capacity; ++i)
  arrNew [i] = _arr [i];
for(; i < newSize; ++i)
  arrNew [i] = _valDefault;
_capacity = newSize;
  //释放旧内存
  delete []_arr;
  // 用新数组替换旧数组
  _arr = arrNew;
}
```

9.1.1 动态 MultiNode

现在该使用动态数组模板来体验它带来的简单性了。让我们从类 MultiNode 开始。在原来受限制的实现代码中，它有两个数组：一个为 Node 的指针数组，另一个为布尔标志数组。第一步将这些数组的类型分别更改成 DynArray<Node*>和 DynArray<bool>。我们不得不在 MultiNode 构造函数的前导中给这些数组的构造函数传递缺省值。访问数组的方法将照常工作（因为重载了 operator []），只是在检查数组边界的地方作一些改动。在这些地方，我们要扩展数组，所以，我们必须使用 Set 方法。这一地点正好在 AddChild 方法之内，相应的转换很简单。

```
class MultiNode: public Node
{
public:
  MultiNode(Node * pNode)
       : _aChild(0),
       _aIsPositive(false),
       _iCur(0)
  {
    AddChild(pNode, true);
  }
  ~MultiNode();
  void AddChild(Node * pNode, bool isPositive)
  {
    _aChild.Set(_iCur, pNode);
    _aIsPositive.Set(_iCur, isPositive);
    ++_iCur;
  }
protected:
  int                  _iCur;
  DynArray<Node*>      _aChild;
  DynArray<bool>       _aIsPositive;
```

```
};
MultiNode::~MultiNode()
{
  for(int i = 0; i < _iCur; ++i)
    delete _aChild [i];
}
```

让我们再看一下 SumNode 的 Calc 方法。除了去掉错误检查之外（我们已经去掉了没有必要的标记_isError），工作过程中好像什么也没有修改一样。

```
double SumNode::Calc()const
{
  double sum = 0.0;
  for(int i = 0; i < _iCur; ++i)
  {
     double val = _aChild [i]->Calc();
     if(_aIsPositive[i])
       sum += val;
     else
       sum -= val;
  }
  return sum;
}
```

唯一的区别是，当我们访问数组_aChild [i]和_aIsPositive [i]时，我们实际上调用的是各自动态数组的重载运算符[]。而且，因为方法 Calc 为常量，它是我们所调用的重载版本。

为了测试动态数组，将 initDynArray 设置为 1（在此程序中设置为 0 将不会工作）。

9.1.2　动态 StringBuffer



Script 22

我们可以使用相同的技术使字符串缓冲器动态化。不过，因为字符串缓冲器的增长模式有所不同，所以不能使用一个一般目的性的动态数组，我们将做一些更有针对性的工作。字符串缓冲器需要扩展的唯一场合就是当给它添加新字符串的时候。下面是新的实现代码：

```
class StringBuffer
{
public:
  StringBuffer()
    : _buf(0), _bufSize(0), _curOffset(0)
  {}
  ~StringBuffer()
  {
    delete []_buf;
  }
  int Add(char const * str);
```

```
int GetOffset()const
{
  return _curOffset;
}
bool IsEqual(int offStr, char const * str)const;
char const * GetString(int offStr)const
{
  assert(offStr < _curOffset);
  return &_buf [offStr];
}
private:
  void Reallocate(int addLen);

  int    _bufSize;
  char  * _buf;
  int    _curOffset;
};
```

当缓冲器空间用完时，Add 方法就会重新分配整个缓冲器。

```
int StringBuffer::Add(char const * str)
{
  int len = std::strlen(str);
  int offset = _curOffset;
  //空间是否足够?
  if(_curOffset + len + 1 > _bufSize)
  {
    Reallocate(len + 1);
  }
  // 在这里复制字符串
  std::strncpy(&_buf [_curOffset], str, len);
  // 计算新偏移
  _curOffset += len;
  _buf [_curOffset] = 0;  // Null 符终止
  ++_curOffset;
  return offset;
}
```

重新分配遵循标准的加倍模式，但是要确保新字符串的适合，即使是大小加倍的缓冲器并不足够也应如此。

```
void StringBuffer::Reallocate(int addLen)
{
  int newSize = _bufSize * 2;
  if(newSize < _curOffset + addLen)
    newSize = _curOffset + addLen;
  char * newBuf = new char [newSize];
  for(int i = 0; i < _curOffset; ++i)
    newBuf [i] = _buf [i];
```

```
  delete []_buf;
  _buf = newBuf;
  _bufSize = newSize;
}
```

9.1.3 动态 SymbolTable

除了 StringBuffer 之外，SymbolTable 类的另一部分需要动态化的是偏移数组_aOffStr。

```
class SymbolTable
{
  enum { hTabSize = 65 };
public:
  enum { idNotFound = -1} ;
  SymbolTable();
  int ForceAdd(char const * str);
  int Find(char const * str)const;
  char const * GetString(int id)const;
private:
  HTable         _htab;   // String -> ids
  DynArray<int>   _offStr; // id -> offset
  int            _curId;
  StringBuffer    _strBuf; // Offset -> string
};
```

请注意，我们不再需要析构函数，子对象的析构函数将负责清理工作。

下面是经过改进后的 ForceAdd 实现代码。我们不必再担心偏移表或字符串缓冲器的溢出。

```
int SymbolTable::ForceAdd(char const * str)
{
  int offset = _strBuf.Add(str);
  _offStr.Set(_curId, offset);
  _htab.Add(str, _curId);
  return _curId++; // 后自增
}
```

能够简化一些代码总是令人愉快——这是我最喜欢的编程部分。现在的 ForceAdd 代码阅读起来就像在朗读日本的俳句一样简洁明朗。

下面是符号表的构造缩减的方式：

```
SymbolTable::SymbolTable()
  : _curId(0),
  _htab(htabSize),
  _offStr(0)
{}
```

现在解析一个符号变量变得多么容易，如下所示：

```
// Factor := Ident
if(id == SymbolTable::idNotFound)
```

```
    id = _symTab.ForceAdd(strSymbol);
assert(id != SymbolTable::idNotFound);
pNode = new VarNode(id, _store);
```

我通过断言作为阻止别人更改 SymbolTable 的实现代码来使用一个非动态数组。

9.1.4 动态 Store

Script 23

类 Store 需要两个动态数组以及在检查边界的地方作若干少量的修改。

```
class Store
{
private:
  enum { stNotInit, stInit };
public:
  explicit Store(SymbolTable & symTab);
  bool IsInit(int id)const
  {
    assert(id >= 0);
    return _status.InRange(id)&&
          _status [id] != stNotInit;
  }
double Value(int id)const
{
  assert(id >= 0);
  assert(IsInit(id));
  return _cell [id];
}
void SetValue(int id, double val)
{
  assert(id >= 0);
  if(_cell.InRange(id))
  {
    _cell [id] = val;
    _status [id] = stInit;
  }
  else
  {
    AddValue(id, val);
  }
}

void AddValue(int id, double val)
{
  assert(id >= 0);
```

```
    _cell.Set(id, val);
    _status.Set(id, stInit);
  }
private:
  DynArray<double>       _cell;
  DynArray<unsigned char> _status;
};
```

在 Store 构造函数的前导中，我们给动态数组的构造函数传递参数。内存单元被初始化为 0，并将其状态初始化为 stNotInit。

```
Store::Store(SymbolTable & symTab)

  : _cell(0.0), _status(stNotInit)
{
  // 添加预定义常量
  // 注意: 如果需要更多，则做更一般性的工作
  std::cerr << "e = " << std::exp(1)<< std::endl;
  int id = symTab.ForceAdd("e");
  AddValue(id, std::exp(1));
  std::cerr << "pi = " << 2*std::acos(0.0)<< std::endl;
  id = symTab.ForceAdd("pi");
  AddValue(id, 2.0 * std::acos(0.0));
}
```

最后，我们去掉了一些内在的限制，它反映在 main 中的一些细微简化中。我们不再需要给构造函数传递大小参数。

```
int main()
{
  ...
  SymbolTable symTab;
  Function::Table funTab(symTab);
  Store store(symTab);
  ...
}
```

9.2　标准矢量

Script 24

你已经知道了动态数组的工作方式，下面再介绍 C++标准库中最有用的成员：矢量（vector）。我们期望在动态数组中运用矢量。让我们将 DynArray 出现的所有地方都用 std::vector 替换（是的，它是 std 名字空间的一个成员），确保我们包含了相应头：

```
#include <vector>
```

让我们从 MultiNode 及它的两个动态数组开始。

```
std::vector<Node*> _aChild;
std::vector<bool>  _aIsPositive;
```

在代码可编译之前，我们必须作几个调整。例如，std::vector 没有 Add 方法。但是，我们可以通过调用 push_back 添加新元素，如果有需要它还有扩展数组的附加功能（注意 C++标准库的特殊命名约定。所有的名字都是小写字母，用下划线作单词间的分隔符。这样就可以将它与我们自己起的名字区分开来）。

另一件事：为矢量元素提供缺省值似乎没有简单的途径。幸运的是，C++标准库使用缺省构造函数来填充矢量及其他容器中的空区域。C++还为所有的内置类型提供了缺省构造函数。例如，int 或指针的缺省值为 0，bool 类型的缺省值为 false 等。你可以试一试！

```
int i = int();
bool b = bool();
typedef void * voidptr;
void * p = voidptr();
std::cout << "int " << i << ", bool " << b
          << ", pointer " << p << std::endl;
```

事实表明，std::vector 的引入进一步简化了类 MultiNode 的实现：

```
class MultiNode: public Node
{
public:
  MultiNode(Node * pNode)
  {
    AddChild(pNode, true);
  }
  ~MultiNode();
  void AddChild(Node * pNode, bool isPositive)
  {
    _aChild.push_back(pNode);
    _aIsPositive.push_back(isPositive);
  }
protected:
  std::vector<Node*> _aChild;
  std::vector<bool>  _aIsPositive;
};
```

我之所以删除了成员变量_iCur，是因为不再需要它。我们可以通过调用矢量的 size 方法迅速找出有多少个子代。知道矢量的大小后，我们就可以循环遍历它的元素。不过，如果我们只是需要遍历一个矢量的元素，就像 MultiNode 的构造函数所做的一样，那就可以使用一个更好的、更标准的办法——使用迭代器。

如果你认为矢量是一个数组，迭代器就像一个指向其中元素的指针一样。还记得我说过，最好使用一个索引代替指针来访问数组元素？矢量与之有些相反，至少目前用于迭代是如此。

你可以通过调用矢量的 begin 方法得到一个指向矢量起始处的迭代器。同样，通过调用它的 end 方法可以得到一个指向矢量最后元素之上元素的迭代器。你可以应用自增运算符++

移动迭代器，让它指向矢量的下一个元素。使用相等或不相等运算符可以测试两个迭代器是否相等（即是否指向矢量的中的同一个元素）。最后，使用*（星号）运算符反引用迭代器得到矢量的元素值。

下面是 MultiNode 的新析构函数：

```
MultiNode::~MultiNode()
{
  typedef std::vector<Node *>::iterator NodeIter;
  for(NodeIter it = _aChild.begin();
      it != _aChild.end(); ++it)
  {
    delete *it;
  }
}
```

类型 iterator 在内部是类 vector 的模板。它能导致一个非常笨拙的类型名：std::vector<Node *>::iterator。习惯上为这种复杂的类型名声明一个本地 typedef。

可能你觉得奇怪，vector<T>::iterator 十有八九像按一个指针的方式来实现的。由于这样或那样的原因，如果它不是的话，你仍然可以完成一些通过指针也可以完成的事。更重要的是，它获得了与指针同样的性能。你可以把它像指针一样传递。你可以像指针一样从某个函数返回它。事实上，从性能上讲，上面的代码与下面的代码没有区别：

```
Node * aChild [size];
typedef Node * NodeIter;
for(NodeIter it = aChild; it != aChild + size; ++it)
  delete *it;
```

我之所以强调这一点，是因为一些程序员使用 C++标准库的功能时可能会犹豫不决，他们担心性能的降低。无需担心——C++标准库的创建者首当其冲保证的是其性能。譬如，这就是他们为什么选择以类似指针的方式来迭代，而不应用更安全的序列生成器的原因。

类似指针的迭代需要两个单独的数据块——当前位置及结束位置。一个序列生成器两者均可访问。通过序列生成器，不会有错误地比较某矢量当前位置与另一个矢量结束位置的危险。事实上，如果你担心这类错误的发生，你可为两个迭代器创建一个序列生成器。

```
class NodeSeq
{
public:
  NodeSeq(std::vector<Node *> & vec)
    : _cur(vec.begin()), _end(vec.end())
  {}
  bool AtEnd()const { return _cur == _end; }
  void Advance(){ ++_cur; }
  Node * Get(){ return *_cur; }
private:
  std::vector<Node *>::iterator _cur;
  std::vector<Node *>::iterator _end;
};
```

序列生成器的唯一缺陷是它按值传递的开销稍大一些——你可以看到，序列生成器的开销通常是对应迭代器开销的两倍。而且一旦你使用了标准算法，迭代器就会被多次传递。



Script 25

下一个动态数组的替换是符号表的_offStr 数组。我们必须重新编写 ForceAdd 方法，附带去掉_curId 数字。

```
std::size_t SymbolTable::ForceAdd(char const * str)
{
  int offset = _strBuf.Add(str);
  std::size_t id = _offStr.size();
  _offStr.push_back(offset);
  _htab.Add(str, id);
  return id;
}
```

你可能还记得，我们使用偏移数组的当前位置作为字符串的 ID。同样，vector 的 size 方法帮助我们定位当前矢量的结尾。但这里有一个小问题。方法 size 返回一个无符号类型（大小毕竟不能为负）。到目前为止，我们一直使用 int（它是有符号的）作为 ID。当你试图隐式转换一个无符号类型为一个有符号类型时，大部分编译器会发出警告，因为可能会丢失精度。在本程序中，这一问题几乎不可能发生，因为只有插入多于 20 亿的标识符时才能发生这种情况（当转换成带符号的 32 位整数时， 20 亿到 40 亿之间的无符号数字变成负数）。不过，由于这种溢出的可能性太遥远了，所以，我们不希望编译程序时看到任何警告。

这就是我决定从使用 int 类型转换到使用库定义类型 std::size_t（可以在<cstdlib>头中找到）的原因。这是一个痛苦的过程——我必须找到所有使用了“int id”的地方，再将它们改成“std::size_t id”。而且，值 idNotFound 不再定义为一个等于-1 的枚举（它是带符号的），而是定义为 SymbolTable 的一个静态常量成员：

```
class SymbolTable
{
public:
  static const std::size_t idNotFound = 0xffffffff;
  ...
};
```

请注意静态常量成员在类的作用域内初始化的方式。值 0xffffffff 是最大的 32 位无符号数。只要你没有在代码中使用它的地址，标准就不要求在实现文件中有单独定义（但这一次没有初始化）。

你注意到编译器没有对 SymbolTable 构造函数中的矢量初始化发出警告了吗？与 DynArray 不一样，std::vector 不带缺省元素值。不过，它有一个单参数构造函数，此构造函数接受矢量的初始大小。当然，这并不是我们所需要的，所以最好删除它。

Script 26

最后，让我们去掉 Store 中的两个 DynArray。处理这种情况需要一些技巧。其中一个数组保存了编译器不知道其缺省值的枚举值。不必使用更复杂的技巧，我们为什么不简单地用一个 bool 矢量替换此数组？我们对此数组所询问的问题是给定元素是否已经初始化，答案为“是”或“没有”。下面是 Store 的新数据成员：

```
std::vector<double>  _cell;
std::vector<bool>    _isInit;
```

下一步，我们必须将所有对 DynArray::InRange 的调用用一个对 vector::size 的测试来代替，例如：

```
bool IsInit(std::size_t id)const
{
  assert(id != idNotFound);
  return id < _isInit.size()&& _isInit [id];
}
```

注意一个重要的变化，为了避免来自编译器的有符号/无符号不匹配警告，我必须作出这样的修改。

AddValue 方法必须能在每个矢量的当前结尾之上插入一个条目。完成的方式是首先调整它们的大小。resize 方法增加了矢量的大小，并用缺省值填充新位置。

```
void AddValue(std::size_t id, double val)
{
  assert(id >= 0);
  _cell.resize(id + 1);
  _isInit.resize(id + 1);
  _cell [id] = val;
  _isInit [id] = true;
}
```

Script 27

小结：任何程序都不能由于使用定长数组而存在不合理的限制。只要需要使用可调整大小的数组，就应该使用 C++标准库矢量。

第10章 资源管理

我不知道你是否注意到，我们的程序有一个重要缺陷：它假定内存分配永远不会失败。这一假设显然是不真实的。假如在一个虚拟内存系统中，分配极限相当高。事实上，计算机甚至没有物理内存上的限制。内存分配失败的唯一的可能性是磁盘上的交换文件占满了所有的空间，这对于庞大的应用程序以及多媒体文件不是没有可能发生。一个工业强度程序应该杜绝这种可能性。问题是如何杜绝呢？当分配内存失败时，程序应该怎样做呢？首先，如果我们不做出任何反应（实际上我在此程序中还没有这样做），系统将意外终止应用程序。这是不能接受的！

稍微令人可接受的办法是，打印一个离开信息并退出程序。对于一个像我们的计算器这样的小程序，这样做是比较合适的。而不能让用户冒着计算器意外退出时白费数小时工作的危险。

还有几种不同的途径来全局中途截取内存分配失败。一种途径是捕获 std::bad_alloc 异常（稍后再详细讨论之）；另一种途径是创建自己的“new 处理器”并在程序启动时通过运行时注册它。这一函数 set_new_handler 在头<new>中定义。它接受一个函数指针，当 new 失败时就调用这一函数。下面举一个用户自定义 new 处理器的例子：

```
void NewHandler()
{
  std::cerr << "Out of memory\n";
  exit(1);
}
```

它通过调用一个特殊的函数显示一条信息并从<cstdlib>退出。

注册处理器的方式——最好在 main 的起始处：

```
set_new_handler(&NewHandler);
```

如果你使用的是一个非标准兼容性编译器，你可能要不得不修改此代码。例如，在 Microsoft VC++ 6.0 中，有一个名叫_set_new_handler 的函数起着相同的作用。它的声明放在<new.h>中。你还必须修改传递给 int NewHandler（size_t size）的函数信号。

尽管在许多情况下，这一解决方案是完全可以接受的，但大部分商业程序要求有更成熟的技术。试想一下如果一个字处理程序或一个编译器突然退出，而你没有丝毫的机会来保存工作的糟糕情形（似乎经历过嘛？！）。

当内存分配失败时会发生什么？在标准 C++中，运算符 new 产生一个异常。因此让我们先讨论一下异常。

10.1　异常

在 C++中，我使用异常来处理意外的失败。实质上，什么是异常、什么不是异常是由程序员决定的。内存分配失败被看作异常，就像磁盘读或写失败一样。另一方面，“未找到文件”或“至文件结尾”被看作是正常事件。当发生一个异常条件时，程序会抛出一个异常。抛出越过正常的流程并进入最近 try/catch 块的 catch 部分。所以，抛出的行为实际上就像一个非本地 goto 一样（“非本地”的意思是流程控制可以跳过函数调用）。使用 goto 来控制流程是尽量要避免的办法，因为它极容易出错，而且给程序的维护带来巨大的困难。我们将在 10.2 一节列举这样的一个例子）。

为了更好地理解异常的使用，让我们回顾一下计算器程序。我们可以预见的惟一异常条件就是内存分配失败。正如我以前所提过的，当使用 new 不能分配所需的内存时，它就会抛出一个 std::bad_alloc 类型的异常。

写本书时，许多编译器仍然没有标准化。但是不要失望，你可以编写自己的抛出异常的 new 失败处理器来解决这一问题。

```
class bad_alloc {};
int NewHandler(size_t size)
{
  throw bad_alloc;
  return 0;
}
```

如果你使用的是 Microsoft 6.0 编译器，在 main 中首先调用 _set_new_handler (&NewHandler)。注意，处理器通过它的缺省构造函数创建一个 bad_alloc 类型对象，而它没有任何其他作用。类型 bad_alloc 本身是一个哑元——我们定义它只是为了能区分不同的异常类型。

Script 28

如果你没有捕获这一异常，程序将会突然终止。所以，最好还是要捕获这一异常。本程序将在流程控制进入 try/catch 块的时候立即捕获异常。让我们在 main 中打开一个 try 块。在

我们构造符号表之前必须完成这一点，符号表的构造函数进行首次内存分配。在 try 块的结尾，我们必须放置一个 catch 块，它的作用是捕获异常。

```
int main()
{
  try
  {
    char buf [maxBuf];
    Status status;
    SymbolTable symTab;
    Function::Table funTab(symTab);
    Store store(symTab);
    ...
  }
  catch(std::bad_alloc)
  {
    std::cerr << "Out of memory!\n";
  }
  catch(...)
  {
    std::cerr << "Internal error\n";
  }
}
```

在这里，我们有两个 catch 块，它们相邻放置。第一个只捕获 bad_alloc 类型异常；第二个（带有省略号者）将捕获其他所有异常。程序中的“内部错误”可能是一般保护性错误或除 0 的错误——倒不是我们想要允许这样的漏洞存在，但显示相关信息后再退出总比系统意外终止并显示无关信息要好。

try/catch 块总是放在程序的 main 函数体中。

图 10-1 演示了在 Parser::Expr 内分配 SumNode 过程中抛出异常后的流程控制，Parser::Expr 由 Parser::Parse 调用，而 Parser::Parse 从 main 内的 try 块中被调用。

通过越过正常的流程控制，异常的使用解决了错误的传播问题（参见图 10-1 中的矩形框）。

在以前，new 常常报告内存分配失败并返回一个空指针。为了防止灾难性崩溃的出现，程序员必须测试每次分配的结果。这样做不仅冗长、易出错，而且当发生分配失败时，大部分时间并不能立即清楚该做什么。所有的低级子程序都可能停止执行并返回调用子程序错误的代码。当然，这就把检查错误的担子交给了调用者。在大部分情况下，调用者选择余地很少，只能将相同的错误代码传递给它的调用者，依此类推。最后，在某个较高层——也许在 main 中，程序员对此问题提供了处理代码，例如，报告给用户、尝试补救一些工作，也许是退出程序。这种方法不仅易出错（忘记检查错误返回），而且它极易造成 main 流程控制的混乱。

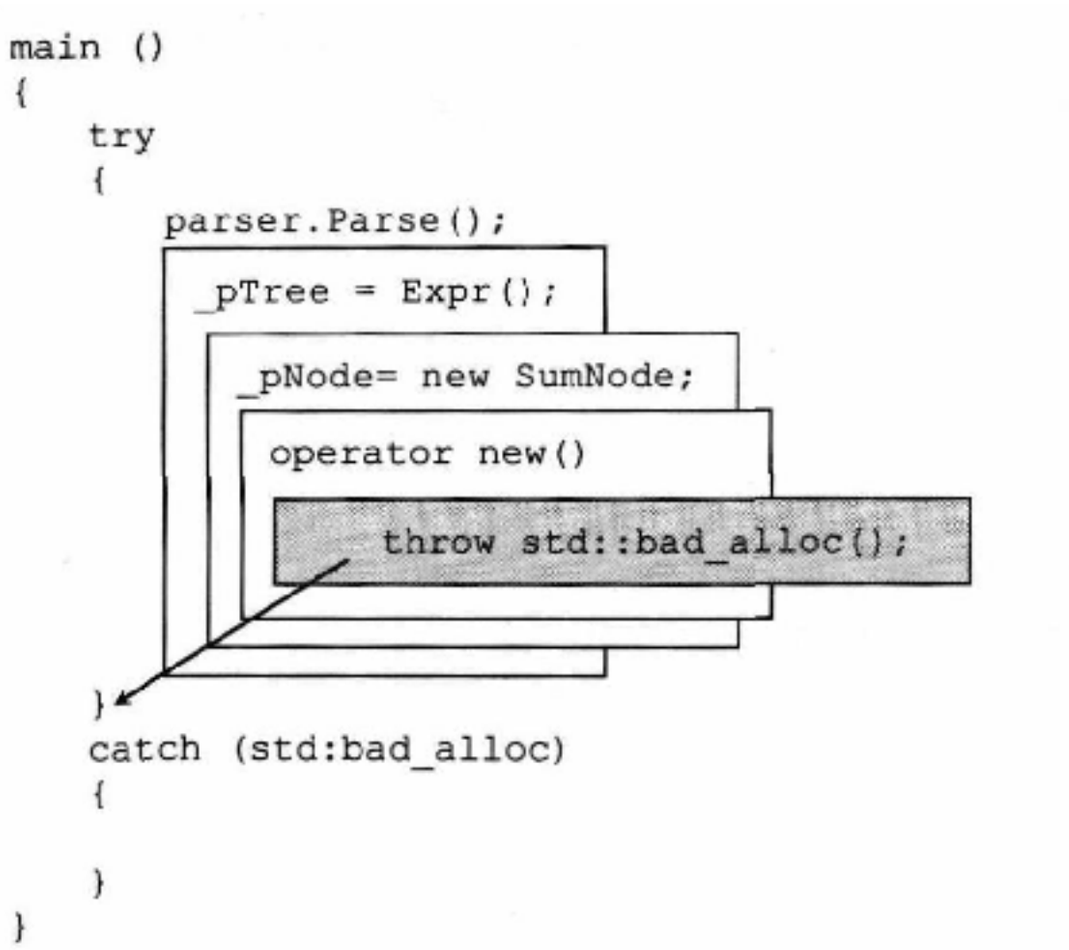


图 10-1　多重函数调用间的 throw 和 catch

10.2　堆栈展开

针对传统的错误传播，人们发明了许多清理方案。许多过程先分配然后再释放内存。当发现一个错误时，在执行一个严重错误返回前，无论内存中分配的内容是什么，它都必须被释放。因为在许多情况下，内存资源在过程的底端释放，只要发现一个错误，对应的方法就用 goto 跳转到其底端。没有错误传播的典型过程如下所示：

```
void foo()
{
  Type1 * p1 = new Type1;
  call1(p1);
  Type2 * p2 = new Type2;
  call2(p2);
  delete p2;
  delete p1;
}
```

错误传播及其清理如下所示：

```
ERRCODE foo()
{
  ERRCODE status = errNoError;
  Type1 * p1 = 0;
  Type2 * p2 = 0;
  p1 = new Type1;
  if(p1 == 0)
  {
    status = errOutOfMemory;
```

```
  goto Cleanup;
}
status = call1(p1);
if(status != errNoError)
  goto Cleanup;
p2 = new Type2;
if(p2 == 0)
{
  status = errOutOfMemory;
  goto Cleanup;
}
status = call2(p2);
if(status != errNoError)
  goto Cleanup;
// 更多相同情况
Cleanup:
delete p2;
delete p1;
return status;
}
```

这里的 Cleanup 是一个用作 goto 跳转目标的标签。所有这些代码只是为了分配和销毁两个对象并调用两个函数。

如果你认为我夸大其词，那就错了！我见过许许多多这样编写的商业代码。顺便要说的是，如果有人在这一过程的中间添加一个条 return 语句，整个清理逻辑过程将彻底失效。

稍等，在完全解决问题前还必须解决其他一些麻烦！使用异常来传播错误不会立即解决这一问题。你没有机会来清理分配，除非你在每个过程中放置 try/catch 对，看起来似乎也应该如此。将以上例子直接转换成如下代码：

```
void foo()
{
  Type1 * p1 = 0;
  Type2 * p2 = 0;
  Try
  {
    p1 = new Type1;
    call1(p1);
    p2 = new Type2;
    call2(p2);
    // 更多相同情形
  }
catch(...)
{
  delete p2;
  delete p1;
  throw;
```

```
    }
    delete p2;
    delete p1;
}
```

当在 catch 中捕获到同样的异常时，throw 就会再次抛出此异常。

代码看起来较为清晰了一些，但它仍然容易出错。在此过程的开始将指针设置为 0，在中途赋值，并在结尾释放（注意其中两个 delete 语句，它们看起来不顺眼，但是必需的）。容易出错的地方仍然很多。此时，你可能感到失望了，你开始想借助于 Java 或其他一些提供了自动垃圾回收的语言，但在真正下定决心前，请继续阅读下面的内容。

C++中有一种正是用于清理的机制：自动对象的构造函数和析构函数对。一旦在某个本地作用域内构建了一个对象，当流程控制离开这一作用域时，这一对象将被自动销毁。重要的是，它与如何退出作用域无关：流程控制可以自然到达闭合括号，或者通过一个 goto、break 或 return 语句跳出作用域。自动对象总要被销毁。是的——这非常重要，由于异常导致离开作用域时，自动对象也会进行清理！请再看图 10-1——在外部 try 和内部 throw 块之间所有嵌套作用域中的所有基于堆栈的对象都被自动销毁，而且它们的析构函数按构建的反顺序执行。

稍后我将演示如何应用这一强大的机制（称做堆栈展开）来为我们完成所有的清理工作。不过，首先让我们看看几种复杂的情况：

例如，当在构建过程中抛出异常时会发生什么？假想有一个包含两个嵌入的对象。在其构造函数中创建这两个嵌入对象，或隐式创建或显式创建（通过前导）。现在，假设第一个嵌入对象构造完毕，但第二个嵌入对象的构造函数抛出一个异常。你认为将发生什么？在展开过程中，第一个嵌入对象应该被销毁吗？没错！正是如此——完全构建的子对象的构造函数在展开时执行。

如果使用 new 来动态分配对象，而且它的构造函数抛出一个异常呢？为此对象分配的内存不就要释放吗？也是对的！如果构造函数抛出一个异常，通过 new 分配的内存将自动释放。

如果堆对象有子对象，而且第 n 个嵌入对象的构造函数抛出一个异常呢？n-1 个子对象不就应该被销毁并释放内存吗？对——正是这样！

还记对象构建规则吗？首先构建基类，然后构建所有嵌入对象，最后才执行构造函数体。如果在任何一个构建阶段抛出了异常，整个过程被逆向进行，并且将所有完全构建的对象销毁（销毁顺序与构建顺序相反）。

如果在构建某数组的第 n 个元素时抛出一个异常呢？所有的 n-1 个完全构建的元素不就要销毁吗？当然，正是如此。

一切正是如此工作的！

这些规则都将发生作用，不管对象（数组）分配于堆栈上还是堆上。唯一的不同之处是，一旦堆对象完全构建，它（以及它的各部分）将不参与堆栈展开过程。

堆栈展开时唯一一种不利的情况是——在析构函数中抛出异常。所以，不要在一个析构

函数中分配一些暂时内存、将文件保存到磁盘，或做其他一些容易出错的动作。析构函数在后台不知不觉地进行清理。

> 决不要在析构函数中执行一些可能抛出异常的动作。

现在再看符号表。

```
class SymbolTable
{
...
private:
  HTable           _htab;
  std::vector<int>  _offStr;
  StringBuffer      _strBuf;
};
```

它有 3 个嵌入对象，它们都有各自的构造函数。哈希表分配一个列表数组，偏移的矢量分配一些初始存储空间，而且字符串缓冲器为字符串分配一些缓冲器。以上所有分配都可能失败并抛出异常。

如果哈希表的构造函数失败，清理就不会进行（除非整个 SymbolTable 对象使用 new 分配——在这种情况下，它的内存将自动释放）。如果哈希表已经成功构造，但矢量构造函数失败了，则在堆栈展开期间执行哈希表的析构函数。最后，如果异常发生在字符串缓冲器的构造函数中，则调用矢量及哈希表的析构函数（调用顺序与此相同）。

哈希表的构造函数本身也可能有一些潜在的失败场合。

```
HTable::HTable(int size)
  : _size(size)
{
  _aList = new List<int> [size];
}
```

为了探讨这一问题，假设 List<int>的构造函数起着一定的作用（例如分配内存）。计算器中构建了 65 个（hTabSize）列表数组。假设第 8 个列表的构造函数抛出一个异常。这将触发堆栈展开，而且自动调用数组前 7 项的析构函数，之后，释放为整个数组分配的内存。另一方面，如果所有 65 个构造函数成功构建，哈希表也成功构建，去掉数组的唯一途径是通过哈希表的析构函数。

但是，因为哈希表是符号表中的一个嵌入对象，而且符号表是 main 中 try 块内的一个自动对象，所以，它的析构可以得到保证。只是幸运地偶然发生吗？不是。让我们再看看 main。

```
int main()
{
  try
  {
    ...
    SymbolTable symTab;
    Function::Table funTab(symTab);
```

```
    Store store(symTab);
    ...
    {
      Scanner scanner(buf);
      Parser  parser(scanner, store, funTab, symTab);
      ...
    }
  }
  catch(...)
  {
    ...
  }
}
```

看起来，与符号表、函数表、存储、扫描器以及解析器相关的所有资源，通过设计确保都得到释放。稍后，我们将看到情况的确如此。

10.3　资源

到目前为止，我们只处理过一种资源——内存。通过调用 new 由运行时系统获取内存，并通过调用 delete 来释放。还有其他一些资源：文件句柄、信号量、引用计数等等（在后面章节的 Windows 编程有关内容中，你将看到大量这样的例子）。它们都有一个共同点——它们必须被分配并最终释放。其定义如下：

> 资源是可以获取和释放的东西。

获取容易实现——难题是释放资源。未释放资源（或资源泄漏）的危害有轻有重，轻者如程序占用比所需内存更多的内存，重者如程序由于未释放信号量而死锁。遵循以下几条分配规则可以保证资源获取和释放的匹配：

> 在构造函数中获取资源；在匹配的析构函数中释放资源。

> 据我所知，这一思想首先由 Margaret Ellis 和 Bjarne Stroustrup 在 *Annotated C++ Reference Manual* 一书中提出。作者讨论异常处理时描述了这一思想（当时尚是试验性的）。在以后的几本书中，Stroustrup 将这一技术称做“资源获取为初始化”。

特别是，通过 new 获取内存只应该在某些对象的构造函数中完成，在析构函数中通过匹配的 delete 释放。在 FunctionTable 和 HTable 的构造函数和析构函数中可以找到这样的匹配对（如果查找<vector>上面的代码，你也能在 in std::vector<T>中找到）。

SymbolTable 没有使用 StringBuffer，而是直接使用字符的数组时，情况也是如此。它的构造函数及析构函数如下所示：

```
SymbolTable::SymbolTable(int size)
  : _size(size),_curId(0),_curStrOff(0),_htab(size+1)
{
  _offStr = new int [size];
  _bufSize = size * 10;
  _strBuf = new char [_bufSize];
}

SymbolTable::~SymbolTable()
{
  delete []_offStr;
  delete []_strBuf;
}
```

遵循了获取规则，但代码并不能做到异常安全性。如果两个（_strBuf 的）分配失败了，请想一想将会发生什么。_offStr 的第一个分配永远也不会被释放——它的内存将发生泄漏。

为了防止编程中处理异常时发生这种情况，我们应该记住下面的资源分配补充规则：

> 资源分配应该是构造函数中的最后一件事，或者在后面放置异常安全代码。

如果在一个构造函数中需要分配多个资源，必须创建子对象来拥有这些资源。作为嵌入对象，即使构造函数在完成前失败，它们也能自动销毁（请参见前一节的讨论）。

你可以看到，不同的讨论得到一个相同的结论——分离子对象是必需的！我们甚至可以完善经验规则“当某个类有太多的数据成员时，建议分离子对象”：当多个数据成员指向同一个资源时应该创建子对象。在本例的符号表中，引入了下面两个子对象来完成这一任务：

```
std::vector<int> _aOffStr;
StringBuffer     _bufStr;
```

10.4 资源所有权

如果一个对象或一个代码块负责释放资源，那它就拥有此资源。资源所有权的概念是资源管理模式中的核心。对象间的 owns-a（拥有成员）关系与更传统的 is-a（继承）关系以及 has-a 关系（嵌入）类似。所有权关系有几种形式。

一个代码块拥有在其作用域内定义的所有自动对象。释放这些资源的任务是完全自动的（调用析构函数）。

所有权的另一种形式就是嵌入。一个对象拥有所有嵌入其中的对象。释放这些资源的任务也是自动完成（外部对象的析构函数调用内部对象的析构函数）。

在两种情况下，资源的生存期限制为嵌入实体的生存期：给定作用域的激活或嵌入对象的生存期。因此，编译器可以完全自动调用析构函数。

对于动态分配对象就不是这样了，它总是通过指针访问。在它们的生存期内，指针可以指向一个资源序列，若干指针可以指向相同的资源。动态分配资源的释放不是自动完成的（也

就是说，C++中没有垃圾回收）。如果对象从一个指针传递到另一个指针，所有权关系就不容易跟踪，程序员常常陷于困境。征兆各不相同：未初始化指针、内存泄漏、重复删除等等。

应用资源分配规则可以澄清所有权关系，而且保证我们列出的大部分错误类型不会出现。代码块只能拥有它的自动对象——这里允许使用“no naked” 指针（没有嵌入对象中）。通过嵌入以及指针，一个对象可以拥有其他对象。在后一种情况中，必须在对象的构造函数中执行资源的获取（指针的初始化），在析构函数中释放。构造函数是纯粹指针惟一可以出现的地方，而且只能存在短暂的时间（参见获取规则的补充条款）。

请注意，如果所有的资源都有所有者，它们的释放就能得到保证。拥有资源的对象或者在全局作用域、本地作用域中定义，或者由其他对象所拥有——通过嵌入或通过构造函数中的动态分配。全局对象在退出 main 时销毁，本地对象在退出它的作用域时销毁，嵌入对象在包装对象销毁时销毁，或在包装构造函数由于异常退出时销毁。因为所有的所有者最终都要被销毁，所以，所有的资源最终都被释放。

你可以有把握地保证不会发生资源泄漏。

10.5　访问资源

代码块或对象可以操作于资源而不拥有它，即不负责它的释放。当授权或传递访问权时，与所有权相反，程序员只要有可能都应该尽量使用引用。我们已经知道，has-access-to 关系可以通过引用得到最好的表达。引用不能表达所有权关系，因为对象不能通过引用来删除（至少不通过一些技巧是不能实现的）。

也存在使用引用很困难或不可能的场合。只是在这种情形下，我们才使用弱指针——不能表达任何所有权关系的指针。在 C++中，强指针和弱指针之间没有语义上的区别，这一区别只是针对具体的代码而言。

has-access-to 关系（通过引用或弱指针）在图 10-2 中用点箭头中来示意。

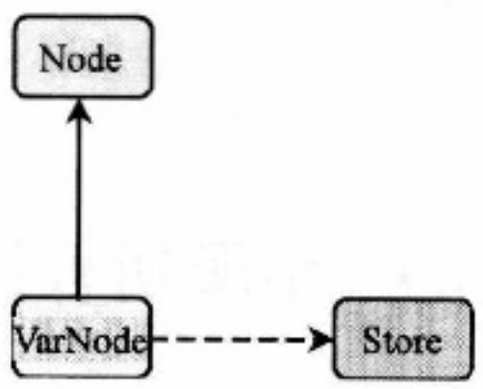


图 10-2　图示 VarNode 和 Store 间的 has-access-to 关系

10.6　智能指针

实现计算器程序时没有考虑资源管理的问题。纯粹指针遍及所有地方，尤其在解析器中。请看下面的代码片段：

```
do
{
  scanner.Accept();
  Node * pRight = Term();
  if(pRight == 0)
```

```
      break;
    pMultiNode->AddChild(pRight,(token == tPlus));
    token = _scanner.Token();
  } while(token == tPlus || token == tMinus);
```

对 Term 调用返回一个临时保存在 pRight 中的节点指针。然后调用 MultiNode 的 AddChild 方法，而且我们很清楚它可能要调整子代数组的大小。如果重新分配失败并抛出一个异常，pRight 指向的树将永远不能释放，因此发生内存泄漏！



Script 29

在演示此问题的系统解决方案前，让我们尝试一件明显的事。因为问题产生于纯粹指针的存在，所以，让我们创建一个特殊用途的类来封装它。这一个类应该在它的构造函数中获取节点，并在析构函数中释放。除此之外，我们希望此类的某个对象像常规指针一样发挥作用。下面是实现的方式。

```
class NodePtr
{
public:
  NodePtr(Node * pNode): _p(pNode){}
  ~NodePtr(){ delete _p; }
  Node * operator->()const { return _p; }
  Node & operator *()const { return *_p; }
  bool IsNull()const { return _p == 0; }
private:
  Node * _p;
};
```

这种对象称做安全或智能指针。通过重载指针访问和指针反引用运算符来实现类似指针的行为。这一聪明的方式使对象的行为像指针一样。特别是，程序员通过“反引用”NodePtr 类型对象可以调用 Node 的所有公有方法（如果有公有数据成员的话，还能访问所有的公有数据成员）。

```
{
  Node * pNode = Expr();
  NodePtr smartNode(pNode);
  double x = smartNode->Calc(); // Pointer-like behavior
  ...
  // 自动析构 pSmartNode
  // pNode 由析构函数删除
}
```

我们通过对 smartNode 应用一个指针访问运算符来调用 Node 的 Calc 方法。因为 smartNode 不是一个指针，编译器从 NodePtr 的定义中选择重载 operator->，并反引用它的返回值（一个指针）。

当然，智能指针本身不能解决解析器中的问题。毕竟，我们不希望通过调用 Term 或 Factor 创建的节点在正常退出其作用域时被销毁。我们希望能在解析树中构建它们，解析树的生存

期扩展到这些方法的本地作用域之外。为了达到这一目的，我们必须放宽资源获取规则。

10.7　所有权转移：第一次尝试

当给定资源的生存期可以映射到一些作用域的生存期时，我们将此资源封装在一个聪明的指针中，而且我们正是这样做的。如果不能这样做，我们必须在作用域间传递资源。这一转移有两个可能的方向：向上或向下。资源可能从一个过程上传到调用者（即返回），或者从调用者下传到过程（作为参数）。我们假定在传递前，资源由某些类型的拥有者拥有（例如一个智能指针）。

下传资源到过程相对较容易。我们可以简单地给所有者对象（在这里是一个智能指针）传递一个引用，并让这一过程获取所有权。我们将给智能指针添加一个特殊的方法 Release 来释放资源的所有权。

```
Node * NodePtr::Release()
{
  Node * tmp = _p;
  _p = 0;
  return tmp;
}
```

注意，Release 把内部指针清零，因此 NodePtr 析构函数中的 delete 不再销毁这一节点。

我们还需要一种方式来给现有的 NodePtr 传递资源所有权。

```
void Reset(Node * pNode)
{
  delete _p;
  _p = pNode;
}
```

上传资源较为麻烦一些。从技术上讲，这是完全可以实现的。我们必须调用 Release 从所有者获取资源，然后返回纯粹的资源。例如，下面是我们从 Parser::Expr 返回一个节点的方式：

```
Node * Parser::Expr()
{
  // 解析一个项
  NodePtr pNode(Term());
  ...
  return pNode.Release();
}
```

使问题复杂化的地方是，现在 Expr 的调用者有一个纯粹指针。当然，如果调用者聪明的话，他或她将立即发现这一指针的一个新所有者——估计是一个智能指针——就像我们在前面对 Term 的结果所做的一样。但是，希望程序员特别关注由 new 和 Release 返回的纯粹指针是一回事，但希望对碰巧返回一个指针的每个过程保持同样的警惕完全是另外一回事，因为

不能立即清楚哪个返回强指针（应该被删除），哪个返回弱指针（不应该删除）。

当然，你可以研究每个调用过程的代码发现你所期望找到的内容。你可能希望传递所有权的过程在它们的头文件中作适当注释。否则你要依赖于一些特殊的命名约定——例如用资源返回过程的名字以前缀“Query”开始（的确找到了这样的名字）。

幸运的是，你不必做任何一件这些可怕的事。还有更好的办法，请继续阅读下面的内容！

小结：即使在我们所应用的方法学中存在一些大的漏洞，但我们总算完成了任务。根据资源获取规则，我们封装了所有的资源。这将保证碰到异常时能进行自动清理。我们编写了一个不太完善的方法来在资源所有者之间上传和下传资源。

10.8 所有权转移：第二次尝试

到目前为止，我们已经尝试通过过程边界转移资源来从它的所有者释放资源，按“纯粹”形式来传递它，然后再立即封装它。明显的危险就是，尽管传递在运行程序的十亿分之一秒内发生，释放之后的资源接受代码可能要编写数月、甚至数年。这一过程的两边并不需要对话。

但是有人会问为什么必须释放传递过程中的资源呢？为什么不能连同它的封装器一起传递呢？让我们看看到底该做些什么。

首先，如果我们从过程“原封不动”地给调用者传递智能指针，我们最终将得到一个悬空指针。请看下面的例子：

```
NodePtr Parser::Expr()
{
  // 解析一个项
  NodePtr pNode = Term();          // <- 赋值
  ...
  return pNode;                    // <- 按值
}
NodePtr Parser::Term()
{
  NodePtr pNode = Factor();        // <- 赋值
  ...
  return pNode;                    // <- 按值
}
```

还记得如何从过程返回对象吗？首先，调用拷贝构造函数来构建一个临时对象，然后，调用赋值运算符将这一临时对象复制到调用者变量中。如果对象没有定义它自身的拷贝构造函数（或赋值运算符），则编译器为它提供一个。缺省拷贝构造函数/赋值运算符执行对象的浅层复制。智能指针的缺省复制没有复制它所指向的对象。

这倒不错——我们并不希望得到资源的一个拷贝。问题在于，我们最终得到两个指向同一个资源的指针。这就不合理了。更糟糕的是，其中一个指针超出了作用域——这就是在过

程中定义的智能指针。它将删除它所指向的对象，而并没有意识到它的克隆指针也同时指向这一对象。我们返回了一个智能指针，但它指向一个已经删除的对象——它成了一个悬空指针。

由于某些原因，我们需要强制执行规则：在任何给定时间给定资源只能有一个所有者。如果有人试图克隆一个智能指针，我们必须将资源从原始拥有者转移。如果我们定义自己的拷贝构造函数及赋值运算符，我们就可以实现这一点。

```
NodePtr::NodePtr(NodePtr & pSource)
  : _p(pSource.Release())
{}

NodePtr & NodePtr::operator=(NodePtr & pSource)
{
  if(_p != pSource._p)
  {
    delete _p;
    _p = pSource.Release();
  }
}
```

请注意，这些不是你常用的拷贝构造函数及赋值运算符。它们不带指向资源对象的常量引用。因为它们要修改资源，所以不能这样做。通过将其内容清零来进行修改。

我们不能称之为真正的“复制语义”，我们将它称做“转移语义”。我们可以通过赋予一个类“转移构造函数”以及一个重载赋值运算符而赋予它转移语义，这两个操作都从参数转移了资源的所有权。

Script 30

为了将一思想推广到所有的指针类型，我们需要一个模板类——一个转移语义的智能指针。但不是从头开始创建它，我们将使用 C++标准库中已经实现的模板。这一模板叫 auto_ptr，它在头文件<memory>中定义。因此，我们将使用 std::auto_ptr<Node>，而不使用 NodePtr。

> C++标准委员会直到最终文档发布的最后一刻还在为制定 auto_ptr 的定义而操劳[1]。如果你的编译器库所包含的是旧版本库，你应该可以理解。例如，你可能发现 auto_ptr 除了包含一个指针外还包含一个所有权标记。这种实现方式现在已废弃了。

让我们从 UniNode 开始，它必须在构造函数中接受它的子节点的所有权。但是现在，子节点在 auto_ptr 中传递。我们可以通过调用它的 release 方法从 auto_ptr 重新得到它。然后，我们可以保存这一指针并在 UniNode 的析构函数中删除它——资源管理的灵魂。但是，还有更好的办法。我们可以在 UniNode 中嵌入一个 auto_ptr 并让它处理所有权事务。

[1] ISO/IEC 14882: 1998（E）, *Programming Language*—C++。

```
class UniNode: public Node
{
public:
  UniNode(std::auto_ptr<Node> & pChild)
    : _pChild(pChild)// <- “转移”构造函数
  {}
protected:
  std::auto_ptr<Node> _pChild;
};
```

看看发生的一切——因为嵌入对象自动销毁，UniNode 不再需要有析构函数了。而且，我们也不必根据构造函数的参数调用 release 了，因为我们可以直接将它传递给嵌入 auto_ptr 的“转移”构造函数。

这一变化容易下传到 FunNode 和 UMinusNode。例如：

```
class UMinusNode: public UniNode
{
public:
  UMinusNode(std::auto_ptr<Node> & pNode)
    : UniNode(pNode)
  {}
  double Calc()const;
};
```

在内部，UMinusNode::Calc 的实现保持未变，因为访问指针与访问 auto_ptr 的语法相同。

我们可以立即对 BinNode 和 AssignNode 进行相同的转换。BinNode 包含两个 auto_ptr。

```
class BinNode: public Node
{
public:
  BinNode(std::auto_ptr<Node> & pLeft,
          std::auto_ptr<Node> & pRight)
    : _pLeft(pLeft), _pRight(pRight)
  {}
protected:
  std::auto_ptr<Node> _pLeft;
  std::auto_ptr<Node> _pRight;
};

class AssignNode : public BinNode
{
public:
  AssignNode(std::auto_ptr<Node> & pLeft,
             std::auto_ptr<Node> & pRight)
    : BinNode(pLeft, pRight)
  {
    assert(_pLeft->IsLvalue());
```

```
    }
    double Calc()const;
};
```

与以前一样，我们不再需要显式析构函数。

我们可以继续使用解析器本身的析构函数。它可以通过 auto_ptr 拥有自己的解析树，而且去掉它的显式析构函数。

现在，让我们看看添加了 auto_ptr 后的 Parser::Expr 方法。请注意我们是如何通过调用方法 get 访问保存在 auto_ptr 内部的指针的。

```
std::auto_ptr<Node> Parser::Expr()
{
  std::auto_ptr<Node> pNode = Term();
  if(pNode.get()== 0)
     return pNode;

  EToken token = _scanner.Token();
  if(token == tPlus || token == tMinus)
  {
    // Expr := Term {('+' | '-')Term }
    std::auto_ptr<MultiNode> pMultiNode(
        new SumNode(pNode));
    do
    {
      _scanner.Accept();
      std::auto_ptr<Node> pRight = Term();
      if(pRight.get()== 0)
        return pNode;
      pMultiNode->AddChild(pRight,(token == tPlus));
      token = _scanner.Token();
    } while(token == tPlus || token == tMinus);
    // 成员模板支持
    pNode = pMultiNode; // <- 提升转换!
  }
  else if(token == tAssign)
  {
    // Expr := Term = Expr
    _scanner.Accept();
    std::auto_ptr<Node> pRight = Expr();
    if(pRight.get()== 0)
      return pNode;
    // 假定项为左值
    if(pNode->IsLvalue())
    {
      // 赋值节点
      pNode = std::auto_ptr<Node>(
              new AssignNode(pNode, pRight));
```

```
    }
    else
    {
      _status = stError;
      pNode = Expr();
    }
  }
  return pNode;
}
```

在这一代码中有一个地方需要加以解释。我们处理的是两种 auto_ptr 类型——一种是封装 Node，另一种是封装 MultiNode（只有 MultiNode 才有 AddChild 方法）。在某些时候，当我们添加子代时，我们必须将一个指向 MultiNode 的指针转换成指向 Node 的指针。对于指针，这可以顺利完成，因为 MultiNode 与 Node 间具有“is-a”关系（它们通过继承关联）。但是 auto_ptr<Node>与 auto_ptr<MultiNode>是完全不同的类。它们是模板 auto_ptr 的两个单独实例。所以下面这行代码：

```
pNode = pMultiNode; // <- 提升型类型转换!
```

隐藏了一些相当有趣的细节。事实上，有一种特殊机制称做成员模板，它可以用于 auto_ptr 内部使得这些转换类型（称做提升转换，因为它将指针转换到类层次的上面级别）的完成变成可能。看一下如下的代码——一个新的转移构造函数以及一个新的赋值运算符添加到 auto_ptr 模板上[1]。

```
template<class T>
class auto_ptr
{
public:
  ...
  // 从派生类“转移”
  template<class U>
  auto_ptr(auto_ptr<U> & pSrc)
  {
    _p = pSrc.release();
  }
  // 派生类的赋值
  template<class U>
  auto_ptr & operator=(auto_ptr<U> & pSrc)
  {
    if(this != &pSrc)
      _p = pSrc.release();
    return *this;
  }
};
```

[1] 实际的库实现代码更复杂。因为转移语义要求非常量引用参数，所以这一简单的实现代码对于临时 auto_ptr 对象不会发挥作用（不能带有指向临时对象的非常量引用）。例如，当按值从某过程返回 auto_ptr 时创建这一临时对象。C++标准库使用的技巧是引用一个中间类 auto_ptr_ref。但这已超出了我们的讨论范围。

这些新方法本身就是模板。它们由另一个类型 U 参数化，U 与参数化 auto_ptr 的类型 T 不同。

新转移构造函数可以从一种类型的 auto_ptr（由 U 参数化）构建另一种类型的 auto_ptr（由 T 来参数化）。事实上，这一构造函数是一个由两种类型（T 和 U）参数化的模板。

新赋值运算符重载可以接受由任意类型 U 参数化的 auto_ptr。在下面语句中正是调用了这一覆盖：

```
pNode = pMultiNode; // 从 MultiNode 向上转换到 Node
```

这里，pNode 为 auto_ptr<Node>类型，pMultiNode 为 auto_ptr<MultiNode>类型。编译器将自动尝试实例化 auto_ptr<Node>::operator=（auto_ptr<MultiNode> &）的适当实现方法。然而这里是类型安全受到威胁的地方。当编译器根据 operator=模板生成下面代码时：

```
_p = pSrc.release();
```

它必须把从 pSrc.release 返回的、指向 MultiNode 的指针赋值给指向 Node 的指针_p。因为两个类之间存在 is-a 关系，所以能进行指针转换。假如我们试图进行相反的转换：

```
pMultiNode = pNode; // <- 错误！向下转换
```

编译器将产生一个错误，因为它在实例化 operator=模板过程中，不能将 Node*转换成 MultiNode*。

其底线是：只有针对指针赋值能工作的类型，auto_ptr 的赋值才会起作用。注意类型不必通过继承关联。例如，既然你可以隐式将一个指向 Node 的指针转换成一个空指针，那么，你也可以将 auto_ptr<Node> 转换成 auto_ptr<void>。

如果你的编译器不支持成员模板，不要失望!你可以使用另外一种方式。既然你不能隐式实现 auto_ptr 转换，那你就必须进行显式转换。

```
pNode = up_cast<Node>(pMultiNode);
```

我创建了向上转换函数模板，由两种类型进行参数化：To 和 From。

```
template<class To, class From>
std::auto_ptr<To> up_cast(std::auto_ptr<From> & from)
{
  return std::auto_ptr<To>(from.release());
}
```

请注意，当我调用这一模板时，我不必指定第二个类型 From。编译器能通过查看参数（在这里是 pMultiNode）的类型而知晓。像以前一样，当为所要求类型对实例模板时，由编译器完成类型检查。

将剩下的 Parser 转换成使用 auto_ptr 相当简单。只有几个地方需要注意。

你不能直接测试 auto_ptr 是否为 0。你必须使用它的 get 方法从 auto_ptr 获取指针。

必须在 Parser.h 中包含 Tree.h（否则，将得到编译器警告）。原因是，这里不再存在显式 Parser 析构函数，所以，编译器不得不提供一个缺省析构函数。但是因为 Parser 包含指向 Node

的 auto_ptr，Parser 的缺省析构函数必须调用 auto_ptr 的析构函数，反过来，auto_ptr 的析构函数要删除 Node。此时，编译器需要知道为 Node 定义的是哪一种析构函数。只有在定义 Node 的头文件 Tree.h 中才能找到这一信息。关键的一点是，在扫描文件 Parser.h 时，编译器意识到对缺省 Parser 析构函数的需要，而且这里正是 Tree.h 中的 include 语句需要访问的地方。

Script 31

不要在一个头文件中包含太多的文件，如果这一点很重要（通常也应该注意这一点），那我们可以寻求另一个解决办法。只为 Parser 提供一个空非内联析构函数。编译器在扫描文件 Parser.h 时会看到这一析构函数的声明，因此认为没有必要再创建一个缺省析构函数了。于是，也就没有必要在 Parser.h 中包含 Tree.h。当扫描实现文件 Parser.cpp 时，编译器就会发现（空）析构函数的定义，并将它应用于所有的缺省析构函数调用，包括 auto_ptr::~auto_ptr。只有在这一时刻，编译器才需要查看 Node 的定义。所以，我们将在实现文件中包含 Tree.h，而不必在头文件中包含它。

小结：内存资源可以使用 auto_ptr 从一个点到另一个点安全保持和转移。你可以将现有程序轻松转换成使用资源管理技术。搜索整个项目中使用了 new 调用的地方，确保所有由 new 返回的指针都安全存在于 auto_ptr 中。

10.9 安全容器

我们的程序还并不完善。程序的许多部分仍然不是 100%的资源安全。让我们回到类 MultiNode 这一部分。

```
class MultiNode: public Node
{
public:
  MultiNode(std::auto_ptr<Node> & pNode)
  {
    AddChild(pNode, true);
  }
  ~MultiNode();
  void AddChild(std::auto_ptr<Node> & pNode,
                bool isPositive)
  {
    _aChild.push_back(pNode.release());
    _aIsPositive.push_back(isPositive);
  }
protected:
  std::vector<Node*> _aChild;
  std::vector<bool>  _aIsPositive;
};
```

下面分析一下 AddChild 内发生的一切。我们在 pNode 上调用 release，然后立即对一个

矢量调用 push_back。我们知道，push_back 可能重新分配矢量的内部数组。重新分配包括内存的分配，我们应该明白，这一过程可能会失败。

你发现问题所在了吗？我们刚刚从 pNode 释放的指针将在分配失败中成为一个纯粹指针。这一指针将永远没有被删除机会。因此产生了内存泄漏。

在这里所需要的是，在指针释放前重新分配矢量。我们可以首先进行下面的调用来强制执行潜在的重新分配：

```
_aChild.reserve(_aChild.size()+ 1);
```

但是这一办法不太聪明（而且给程序员增加了负担）。所以，让我们重新构思这一问题。

这里的难点就是指针矢量，它没有所有权的关系。请注意，节点的释放不是在 vector 的析构函数中完成，而是在 MultiNode 的析构函数中显式完成。我们所需要的是一些指针矢量类型，它们真正拥有保存在其中的对象。

我们可以使用 auto_ptr 的矢量吗？

```
std::vector<auto_ptr<Node> > _aChild;
```

注意：两个大于号之间有空格。如果删除这一空格，编译器将会与右移位运算符混淆。它是 C++语法的另一个特色。让我们看看，当我们在这样一个矢量上执行 push_back auto_ptr 时，重新分配在“转移”赋值前必然发生。而且，我们将去掉显式 MultiNode 析构函数，因为矢量的析构函数将销毁它的所有 auto_ptr，反过来，auto_ptr 销毁所有的节点。

虽然从原理上讲，这是一种有效的解决方案，但在实际中，它很难接受。困难在于矢量的接口完全不适合于具有转移语义的保存对象。矢量的实现是假定它能按值传递或返回条目。除了按值 auto_ptr 包含资源转换的情况外，其他情况下均正常。同样，如果你试图遍历这样一个矢量，每次反引用迭代器时都将得到一个资源转移。因此，除非你非常小心，否则你必定在 auto_ptr 的矢量中出现差错。

Script 32

此时，你可能希望我提供一些 C++标准库中的聪明数据结构，它们能完成我们希望完成的任务。不幸的是，C++标准库中没有这样的所有权方面容器。我们必须自己创建一个。幸运的是，我们具备了所有必需的创建工具。那么，让我们从接口开始。

```
template <class T>
class auto_vector
{
public:
  typedef std::size_t size_t;
public:
  explicit auto_vector(size_t capacity = 0);
  ~auto_vector();
  size_t size()const;
  T const * operator [](size_t i)const;
  T * operator [](size_t i);
```

```
  void assign(size_t i, std::auto_ptr<T> & p);
  void assign_direct(size_t i, T * p);
  void push_back(std::auto_ptr<T> & p);
  std::auto_ptr<T> pop_back();
};
```

注意我的防错性设计。我提供过一个数组访问运算符，它将返回一个左值，但是我决定反其道而行之。相反，如果我希望设置一个值，我们必须使用 assign 或 assign_direct 方法之一。我的想法是，资源转移不应该轻易进行，而且它不需要如此频繁地进行——auto_vector 通常使用 push_back 方法填充。

就目前我们所应用的实现逻辑，我们只是简单地使用了一个动态 auto_ptr 数组。

```
template <class T>
class auto_vector
{
  ~auto_vector(){ delete []_arr; }
private:
  void grow(size_t reqCapacity);
  std::auto_ptr<T> *_arr;
  size_t           _capacity;
  size_t           _end;
};
```

grow 方法分配一个较大的 auto_ptr<T>数组，从旧数组转移所有的项，完成交换后删除旧数组。

```
template <class T>
void auto_vector<T>::grow(size_t reqCapacity)
{
  size_t newCapacity = 2 * _capacity;
  if(reqCapacity > newCapacity)
     newCapacity = reqCapacity;
  // 分配新数组
  std::auto_ptr<T> * arrNew =
          new std::auto_ptr<T> [newCapacity];
  // 转移所有项
  for(size_t i = 0; i < _end; ++i)
     arrNew [i] = _arr [i];
  _capacity = newCapacity;
  // 释放旧内存
  delete []_arr;
  // 用新数组替换旧数组
  _arr = arrNew;
}
```

唯一可能调用 grow 的方法是 push_back：

```
void push_back(std::auto_ptr<T> & p)
{
  assert(_end <= _capacity);
```

```
    if(_end == _capacity)
      grow(_end + 1);
    _arr [_end] = p; // 转移
    _end++;
  }
```

auto_vector 的其余实现过程相当简单，因为所有的资源管理复杂性都在 auto_ptr 之中。例如，assign 方法简单使用一条语句通过赋值运算符，既释放旧对象又转移新对象。

```
void assign(size_t i, std::auto_ptr<T> & p)
{
  assert(i < _end);
  _arr [i] = p;
}
```

方法 assign_direct 运用了 auto_ptr 的 reset 方法：

```
void assign_direct(size_t i, T * p)
{
  assert(i < _end);
  _arr [i].reset(ptr);
}
```

pop_back 方法按值返回 auto_ptr，因为它从 auto_vector 转移了所有权。

```
std::auto_ptr<T> pop_back()
{
  assert(_end != 0);
  return _arr [——_end];
}
```

对 auto_vector 的索引访问通过 auto_ptr 的 get 方法来实现，它返回一个弱指针。

```
T * operator [](size_t i)
{
  return _arr [i].get();
}
```

通过新 auto_vector，MultiNode 的实现不仅 100%资源安全，而且更简单。同时，显式析构函数也无用武之地了！

```
class MultiNode: public Node
{
public:
MultiNode(std::auto_ptr<Node> & pNode)
{
 AddChild(pNode, true);
}
void AddChild(std::auto_ptr<Node> & pNode,
                          bool isPositive)
{
  _aChild.push_back(pNode);
  _aIsPositive.push_back(isPositive);
}
```

```
protected:
  auto_vector<Node>  _aChild;
  std::vector<bool>  _aIsPositive;
};
```

我们如何知道何时达到目的了呢？又如何知道我们的代码完全资源安全呢？有一套简单的测试——至少与目前的内存资源有关——它们可以回答这些问题。我们必须搜索代码中所有出现 new 和 release 的地方，并进行如下一系列测试：

- 这是直接转移到 auto_ptr 吗？
- 我们位于一个对象的构造函数中吗？调用的结果立即保存在对象中，在此构造函数中没有出现异常的代码吗？如果是，在这一对象的析构函数中存在对应的 delete 吗？
- 我们位于一个立即将 new 结果赋值给由对象所有的指针的办法中吗？如果是，前一个值删除了吗？

如果以上条件没有一条满足，我们的代码就不是资源安全。注意，所有这些测试都是本地化的。我们不必经过所有的可能执行路径，只是应用邻近的调用（除了偶而一些析构函数外）。不能再比这简单了！

10.10 迭代器

与其他标准容器一样，auto_vector 也需要迭代器。我不会提供整套迭代器——这一工作太繁琐，最好留给库的实现者来做。在计算器中，我们所需要的是一个唯一前向的常量迭代器，我们将在下面定义它。



Script 33

一个迭代器是一个指向数组元素指针的抽象。它也通常作为一个指针来实现，因此，它可以方便地传递和复制。auto_vector 作为一个 auto_ptr 的数组来实现，因此，我应该将它的迭代器作为指向 auto_ptr 的指针来实现才自然。它的创建、传递以及复制开销少。而且，增加这样一个迭代器与增加一个指针一样容易。唯一阻止我们将 auto_vector 迭代器类型定义为一个指向 auto_ptr 指针的是反引用行为。当你反引用指向 auto_ptr 的指针时，你得到一个 auto_ptr 以及它的所有资源转移语义。我们所需要的是一个在反引用时生成一个常规弱指针的迭代器（而且对于常量迭代器生成一个指向常量的弱指针）。可借助运算符来解决！

```
template<class T>
class const_auto_iterator: public
  std::iterator<std::forward_iterator_tag, T const *>
{
public:
  const_auto_iterator(): _pp(0){}
  const_auto_iterator(std::auto_ptr<T> const * p)
```

```
  : _pp(p){}
bool operator !=(const_auto_iterator<T> const & it)const
  { return it._pp != _pp; }
bool operator ==(const_auto_iterator<T> const & it)const
  { return it._pp == _pp; }
const_auto_iterator operator++(int){ return _pp++; }
const_auto_iterator operator++(){ return ++_pp; }
T const * operator *(){ return _pp->get(); }
T const * operator->(){ return _pp->get(); }
private:
  std::auto_ptr<T> const * _pp;
};
```

首先，一个优秀的迭代器应该继承于 std::iterator 模板，它才能传递给需要询问它的容量的不同标准算法。这一模板由一个迭代器标签参数化（在这里就是 forward_iterator_tag），它定义了迭代器的容量以及它的值类型（在这里是指向 const T 的指针）。严格地说，还有更多的参数：例如，差类型——从一个迭代器减去另一个迭代器时得到的类型——但这一参数缺省为 ptrdiff_t。

我们有两个构造函数——一个为缺省，另一个带有指向 const auto_ptr 的指针。我们使用了两个比较运算符。我们应用了两个自增运算符：一个重载后自增，另一个重载前自增运算符。为了区别两者的声明，为后自增运算符的声明提供一个哑元 int 参数（这是另一个 C++ 语法的特色）。

> 在许多场合，前自增运算符的实现比对应的后自增运算符更简单、更有效。它容易实现首先自增，然后再返回它。另一方面，后自增可能复杂一些。从原理上讲，有时需要记住迭代器的原始状态，自增它的内部变量，而且返回一个使用已记住状态创建的迭代器。
>
> 幸运的是，当你处理指针时，编译器为你完成了所有工作，语句 return p++; 正是完成了这一任务。但是尝试对一个链表应用后自增运算符，你就会明白我的意思。顺便提及，这是我为什么偏向于在我们代码中使用前自增运算的原因，尤其是在所有的 for 循环中。

迭代器有两个反引用运算符，它们都返回一个指向常量的弱指针。第二个让你调用通过类 T 实现的方法。

在类 auto_vector 中，我们提供了一个传统的 typedef 及两个方法（begin 及 end）来支持迭代。再次注意结尾迭代器指向数组的最后元素之上的元素。还请注意，在两个方法中，两个指向 auto_ptr 的指针是如何自动转换成 const_iterator 的。这是因为我们没有将 const_auto_iterator 的构造函数限制为显式。

```
typedef const_auto_iterator<T> const_iterator;
const_iterator begin()const { return _arr; }
const_iterator end()const { return _arr + _end; }
```

最后，让我们看一个迭代器可能用于 SumNode 的 Calc 方法中的例子。

```
double SumNode::Calc()const
{
  double sum = 0.0;
  auto_vector<Node>::const_iterator childIt =
      _aChild.begin();
  std::vector<bool>::const_iterator isPosIt =
      _aIsPositive.begin();
  for(; childIt != _aChild.end(); ++childIt, ++isPosIt)
  {
    assert(isPosIt != _aIsPositive.end());
    double val = childIt->Calc();
    if(*isPosIt)
      sum += val;
    else
      sum -= val;
  }
  assert(isPosIt == _aIsPositive.end());
  return sum;
}
```

我用“可能”一词的原因是，使用迭代器的编程方式相对于传统数组索引迭代的优势并不能马上显而易见，尤其当包含两个平行数组时。我必须用逗号序列运算符将两个自增操作在 for 循环头中压缩到一个位置（你是否还记得，用逗号分隔的表达式在序列中进行求值。序列的值等于最后表达式的值）。

另一方面，如果我们使用链表代替矢量来重新实现的话，代码的转换也更容易。不过，这似乎不太可能。

10.11 错误传播



Script 34

既然代码现在是异常安全的，我们应该重新考虑我们的错误处理策略。当我们发现一个语法错误时，我们正在做些什么呢？我们将解析器的状态设置成 stError，而且从所在的解析方法中返回一个空指针。可能所有的语法错误都在 Parser::Factor 最底层发现。不过，Parser::Term 和 Parser::Expr 都必须处理来自一个较低层解析方法的空节点的可能性。事实上，Parser::Factor 本身必须处理对 Expr 的递归调用可能返回一个 null 的可能性。我们的代码中于是要添加一些如下所示的错误传播代码：

```
if(pNode.get()== 0)
    return pNode;
```

只要有错误通过一套嵌套调用被传播的情况发生，我们就应该考虑使用异常来代

替。如果在发现一个语法错误时，我们让 Parser::Factor 抛出一个异常，我们不必担心通过其他解析方法发现和传播空指针。我们要做的是在最高层（即在 Parser::Parse 中）捕获这一异常。

```
class Syntax {};
Status Parser::Parse()
{
  try
  {
    // 表达式中的一切
    _pTree = Expr();
    if(!_scanner.IsDone())
        _status = stError;
  }
  catch(Syntax)
  {
    _status = stError;
  }
  return _status;
}
```

为了传播语法错误，我定义了一个单独的异常类 Syntax。暂时让此类为空，但它的独特类型可以让编译器将它与其他异常类型区分开来。特别是，我们不想在 Parser::Parse 中捕获 bad_alloc 异常，因为我们不知道该做些什么。它们将在 main 中捕获和处理。

下面是一个代码转换的例子，从 Parser::Factor 转换成使用异常进行语法错误报告。请注意，我们不再测试来自 Expr 的空返回（事实上，我们可以断定它不是 null）。

```
if(_scanner.Token()== tLParen)  // 函数调用
{
   _scanner.Accept();           // 接受 '('
   pNode = Expr();
   assert(pNode.get()!= 0);
   if(_scanner.Token()== tRParen)
     _scanner.Accept();         // 接受 ')'
   else
     throw Syntax();
   if(id != SymbolTable::idNotFound
     && id < _funTab.Size())
   {
     pNode = std::auto_ptr<Node>(
       new FunNode(_funTab.GetFun(id), pNode));
   }
   else
   {
```

```
        std::cerr << "Unknown function \"";
        std::cerr << strSymbol << "\"\n";
        throw Syntax();
    }
}
```

10.12 转换到资源管理

现在你应该相信，不遵循本章所描述的资源管理原则的任何程序几乎不可能做到毫无漏洞。那么，如果你已经开发了一个充斥资源管理漏洞的项目，你应该采取什么样的补救措施呢？将整个工程像对待以上程序一样一次性进行转换吗？不必！你可以一步一步来转换。

首先在项目中找出现了 new 的所有地方。如果应用 new 的地方太多，将整个项目分成若干较小区域，然后依次处理每个小区域。

- 对于构造函数中对 new 的调用，确保在析构函数中存在对应的 delete。考虑在类中嵌入 auto_ptr 来保存调用 new 的结果。
- 对于构造函数外部的调用，确保 new 的结果：
 - 立即保存于一个接受所有权的成员变量中，即在析构函数中存在一个匹配的 delete，而且前一个值在赋值前释放。
 - 立即保存于一个 auto_ptr 中。
- 如果你必须在前一步中创建一个本地 auto_ptr，则在所有权链下传或上传前调用它的 release 方法。这是一个临时解决办法，工作量不大。
- 一旦在程序中应用了 new，找出在前面几步中已经引入的所有 release。然后 标出这些资源转移发生的点。
 - 如果资源下传，修改调用过程的信号来接受一个 auto_ptr（通过引用）。如果这一过程必须进一步传递指针的所有权，则调用 release。你将在下一个迭代中再调用 release。
 - 如果资源上传，按值返回 auto_ptr。修正所有的调用者使之接受返回的 auto_ptr。如果它们必须进一步传递指针所有权，则调用 release。
 - 如果资源所有权下传到一个容器，请考虑将此容器转换成 auto_vector。
- 无论什么时候指针被转换成 auto_ptr，编译器将找到所有不再需要的 delete 调用。请删除它们。

这一转换过程为递归性的。每一步，你可能创建一些对 release 的调用，它们必须在后面进行更正。当你确信其余所有对 new 和 release 的调用都是必需的，而且它们直接给成员变量传递所有权，同时我们知道如何来处理它们，此时，我们就可以完成转换了。

除了内存之外，还有许多其他类型的资源。它们通常从操作系统获取，而且必须返回给它。它们同样应该封装在聪明对象中。我们将在后面看到这样的例子。

10.13　结论

Script 35

从一个简单的问题开始：“当内存分配失败时，程序该怎么办？”答案比想象的要复杂得多。我们必须应用异常、堆栈展开、错误传播、资源管理、资源转移、智能指针、自动指针、聪明容器、自动迭代器以及许多 C++的真正高级功能。你可以看到，每种高级功能都是编写可靠程序所必需的。

编写一个在大部分时间能够正常工作的程序是一回事，编写一个能保证一定可靠性的程序是另一回事。当一个互联网浏览器出现错误时，用户会感到烦恼，但他们通常可以重新启动再次尝试。当一个字处理程序或一个电子表格程序发生错误并丢失了工作文档时，问题就要严重得多了。

如果你想了解开发可靠程序的更多知识，你可以阅读附录 B“事务处理”中的内容。

第11章 使用标准模板库

开始应用 C++标准库后，C++编程就变得容易多了。如果我从头开始编写整个计算器，并且充分应用 C++库，编成的代码就会大不一样。尽管如此，但由于一些原因，我们没有这样做。首先，C++标准库是基于模板的——特别是称做标准模板库（STL）的部分，它拥有所有的容器（矢量、列表、队列、映射等）以及算法（迭代、排序、检索等），所以，我必须首先解释什么是模板以及它们能解决什么样的问题。其次，只有先尝试不应用它们进行编程后，你才会知道它们究竟能为我们减少多少工作量。

如果我们从一开始就完全基于 STL 进行开发，得到的计算器将是什么样子？我起码不会实现哈希表。不，在 STL 中没有现存的哈希表——虽然许多编译器开发商确确实实将要在他们的 C++标准库版本中添加哈希表。但是，另一个有用的数据结构可以担任相同的角色。这就是关联数组（associative array），即 STL 中的 std::map。关联数组像一个数组，但它可以通过变量来索引，变量的类型不一定是整数。

你已经知道了如何定义一个字符串数组。你可以通过一个整数（在这里为无符号整数）进行索引并得到一个字符串。一个关联数组可以反其道而行——例如，你可以通过一个字符串索引它，而且它能反馈给你一个整数，如下所示：

```
std::size_t id = assoc ["foo"];
```

这与我们在符号表中的需求很相似。已知标识符的名字，我们希望得到它的整数 ID。

你可以使用 STL 的映射（map）模板实现一个关联数组。一个 map 通过两种类型来参数化——用于索引它的类型，称做键（key），以及值（value）类型。对于键类型只有一个要求——必须可以通过某种途径比较键。实际上，甚至不需要提供一个方法来比较键的相等性或不等性。映射的重要性是能告知某一个键是否小于另一个键。缺省时，映射希望键类型能直接比较（如 int 或 double），或者具有重载小于运算符。

为了达到大部分不恰当的目的，一种天真的实现关联字符串数组的途径是把 char const * 作为键类型，如下所示：

```
std::map<char const *, std::size_t> dictionary;
```

第一个问题是，虽然这里的确为键类型定义了一个运算符“<”，但它完全不合适。请看下面语句：

```
char const * str1 = "sin";
char const * str2 = "sin";
if(str1 < str2)// <- 并不是你所期望的!
   cout << str1 << " is less than " << str2 << endl;
```

比较的结果毫无意义，因为它所比较的是两个指针——也就是说，从数字上比较两个内存地址。这与我们通常所认为的“字符串比较”丝毫没有关系。

不过，在<cstring>中定义的一个特殊函数 strcmp 可以比较表示为字符指针的字符串。如果两个字符串逐字符相等且长度一样，它返回 0。如果第一个字符串按字典顺序在第二个字符串之前，strcmp 返回一个小于 0 的数。同样，如果第一个字符串在第二个之后，strcmp 返回一个大于 0 的数（把它当成相减两个字符串后的意义）。字典顺序式的比较是使用于字典中的一种按字母顺序的比较方式（暂不要考虑本地化语言问题——把这里所说的字符简单地看作数字）。



Script 36

C++标准库提供了一个 map 版本，它带有第三个模板参数——函子类型。函子（或函数对象）是一个定义了函数调用运算符的类。下面是一个满足我们目的的函子：

```
class LessThan
  : public std::binary_function<char const *,
                                char const *, bool>
{
public:
  bool operator()(char const * str1,
                  char const * str2)const
  {
    return std::strcmp(str1, str2)< 0;
  }
};
std::map<char const *, int, LessThan> dictionary;
```

使用函子对象的途径除了在映射中外，还可能将它作为一个函数（或函数指针——两者语法相同）来使用。

```
char const * str1 = "cos";
char const * str2 = "sin";
LessThan lessThan;
if(lessThan(str1, str2))
   cout << str1 << " is less than " << str2 << endl;
```

事实上，一个函子甚至比一个函数指针更有效，因为它可以内联。在上面的例子中，对 lessThan.operator()的隐式调用根据 LessThan 中的声明，它将被内联。

如果你认为这太复杂——为了完成在映射中比较字符串这样一件简单的事，必须预先声明——不错！事实上，它不仅复杂而且不正确。因为我们所作所为“牛头不对马嘴”。“牛头”就是于我们一直用于表示字符串的低层数据结构。C++标准库有一个完美的“马头”可用于此——它称做 string。

以上符号表的简单化实现方式的第二个问题是：键可能从映射中消失。当你在映射中输入一个新关联——一个（键、值）对时，映射必须记住键及其值。如果使用字符指针作为一个键，映射将保存这一指针，但是你却不知道实际的字符究竟保存在哪里以及要保存多久。有人可能通过一个字符串调用映射，这一字符串分配于堆栈上，一旦调用者返回它就会消失。

我们希望映射拥有键！如果不这样的话，我们不能把 auto_ptr 用作键。C++标准库容器对它们的所有数据使用值语义——它们与转换语义的对象相混淆。这是使用 string 的第二个原因。



Script 37

标准 string 不仅具有值语义，而且它有许多有用的方法——其中一个就是小于运算符。它可以让字符串与映射完美匹配。实现一个字符串关联数组并不复杂。所以，让我们先这样做然后再找问题。下面是新符号表的声明：

```
#include <map>
#include <string>

class SymbolTable
{
public:
  static const std::size_t idNotfound = 0xffffffff;
  SymbolTable(): _id(0){}
  std::size_t ForceAdd(char const * str);
  std::size_t Find(char const * str)const;
private:
  std::map<std::string, std::size_t> _dictionary;
  std::size_t _id;
};
```

请注意，因为 map 现在拥有字符串，所以我们不再使用 StringBuffer。

我们仍然在符号表接口中使用了旧的 C 样式 char const *。这是因为我希望向你演示如何在不必一次性重写所有代码的情况下开始转换旧代码。字符串的两种表示可以和平共处。当然，我们必须在两种表示法的边界作一些转换。例如，ForceAdd 方法开始根据 C-字符串参数创建一个标准字符串。然后，它在关联数组中，将对应的整数 ID 保存在这一字符串下。

```
std::size_t SymbolTable::ForceAdd(char const * str)
{
  std::string s(str);
  _dictionary [s] = _id;
  return _id++;
}
```

这里，用一个键通过索引访问 map 的唯一作用是在 map 中创建一个条目（如果不是已经有了的话）。

不要被访问 map 的语法简单性所迷惑。它用来索引一个关联数组（作为一个 STL map 来实现）所花的时间并不固定不变（因为是通过常规数组或哈希表），而是取决于它的大小（元素的数量）。幸运的是，它不是线性的——数组不是逐个元素进行检索。STL 映射的推荐实现途径是通过可以按对数时间进行的平衡树。对数访问对大部分应用程序都是合适的。

还有一个更显式的添加条目办法，这就是通过调用它的 insert 方法。使用一个 STL pair 模板插入一个（键、值）对。例如，可以按这种方式实现 ForceAdd：

```
std::size_t SymbolTable::ForceAdd(char const * str)
{
  std::string s(str);
  std::pair<std::string, std::size_t> p(s, _id);
  _dictionary.insert(p);
  return _id++;
}
```

如果不是因为我们有时想知道这一名字是否已经存在，符号表的 Find 方法通过返回 _dictionary [str]可简单得以实现。现在我们必须使用一种更复杂的途径：

```
std::size_t SymbolTable::Find(char const * str)const
{
  std::map<std::string, std::size_t>::const_iterator it;
  it = _dictionary.find(str);
  if(it != _dictionary.end())
     return it->second;
  return idNotFound;
}
```

map 的 find 方法返回一个迭代器。这一迭代器可以指向字典的结尾或者指向一个（键、值）对。一个（键、值）对有两个意义明确的公有数据成员：first 和 second。

还应注意，我是通过一个 C-类型的字符串调用 find 的。因为标准字符串（std::string）有一个接受 C-字符串的构造函数，所以，这是可行的。这一构造函数可以用于隐式转换。

不存在从字符串到 char const *的反向隐式转换；然而，却存在一个方法 c_str，它从一个字符串读取一个 C-字符串。

Script 38

我们可以隐式转换来进一步简化符号表的实现代码。我们可以更改方法的信号，让它接受字符串而不是 C-字符串。

```
int ForceAdd(std::string const & str);
int Find(std::string const & str)const;
```

无需修改调用者，以上代码就可以正常工作，因为隐式转换在发挥作用。关键在于，两

个方法带有对字符串的常量引用。编译器在调用和传递常量引用时就不会有创建临时字符串的任何麻烦。不过，如果符号表方法需要非常量引用，此时，以上代码就不能正常工作（我们已经提到过，这里有必要重复：编译器将拒绝创建一个指向临时对象的非常量引用）。

传递字符串有一种有趣的办法——按值来传递。

```
int ForceAdd(std::string str);
int Find(std::string str)const;
```

正如我提到的，因为字符串的值语义，以上代码不会正常工作。它们的行为就像每个复制都是深层复制一样。不过，因为内存分配及复制开销相当大，一个优秀的标准字符串实现应使用各种聪明的技巧来避免这些操作（至少要推迟这些操作）。让我们更详细地探讨这一问题。

11.1 引用计数以及即写即复制



string 1

还记得我们在第 5 章“值语义”一节中对值字符串的讨论吗？下面的情况与这一主题稍微有些不同。

```
class StringVal
{
public:
  StringVal(char const * cstr = 0)
    :_buf(0)
  {
    if(cstr != 0)
        Init(cstr);
  }
  StringVal(StringVal const & str)
    :_buf(0)
  {
    if(str.c_str()!= 0)
        Init(str.c_str());
  }
  ~StringVal()
  {
    delete _buf;
  }
  StringVal & operator=(StringVal const & str);
  char const * c_str()const { return _buf; }
  void Upcase();
private:
  void Init(char const * cstr)
  {
```

```
    assert(cstr != 0);
    _buf = new char [std::strlen(cstr)+ 1];
    std::strcpy(_buf, cstr);
  }
private:
  char  * _buf;
};
```

当重载赋值运算符时，我们必须留意几种特殊情况，如源等于 this 或为空源。

```
StringVal & StringVal::operator=(StringVal const & str)
{
  if(this != &str)
  {
      char const * cstr = str.c_str();
      if(cstr == 0)
      {
        delete _buf;
        _buf = 0;
      }
      else
    {
      std::size_t len = std::strlen(cstr);
      if(_buf == 0 || std::strlen(_buf)< len)
      {
        delete _buf;
        Init(cstr);
      }
      else
        std::strcpy(_buf, cstr);
    }
  }
  return *this;
}
```

我添加了 Upcase 方法来演示修改字符串时发生的一切。

```
void StringVal::Upcase()
{
  if(_buf)
  {
    int len = std::strlen(_buf);
    for(int i = 0; i < len; ++i)
        _buf [i] = std::toupper(_buf [i]);
  }
}
```

特别是，我们可以创建同一个字符串的两个拷贝，而且当我们对其中一个拷贝应用 Upcase 方法时，另一个拷贝保持不变。

```
StringVal str1("foo");
StringVal str2(str1); // 复制
str2.Upcase();
std::cout << str1.c_str()<< std::endl;
std::cout << str2.c_str()<< std::endl;
```

从语义上讲，这正是我们所希望的字符串行为。从性能上讲，我们对这一实现途径不太满意。留给读者一个练习：请考虑当 StringVal 按值返回时，进行了多少内存分配以及字符串复制。

```
StringVal ByValue()
{
  StringVal ret("foo");
  return ret;
}
StringVal str;
str = ByValue();
```

为了节省分配和复制的开销，我们可以考虑这样一种方案：同一字符串的多拷贝在内部共享同一代表对象——它们应该有一个指向同一缓冲器的指针。但是，随即又引出了所有权的问题。应该由谁来删除缓冲器？我们由于某些原因必须时刻跟踪缓冲器有多少个共享所有者。那么，当最后一个所有者消失时，我们应该删除缓冲器。

实现这一方案的最好途径是由两个类来分担责任。一个类封装引用计数共享对象——在这里，这一对象也保存了字符串缓冲器。另一个类处理所有权问题——它增加或减少共享对象的引用计数——在这里是“字符串”对象（如图 11-1 所示）。当字符串对象发现共享对象的引用计数减少到 0 时（即没有人再引用此对象），它将删除此对象。

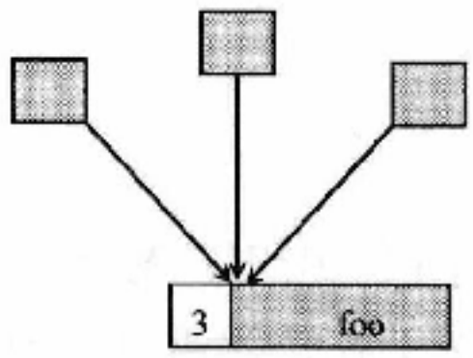


图 11-1 多对象可以共享同一个引用计数。这里有 3 个“字符串”共享同一个引用计数缓冲器

string2

你可以把一个引用计数当成一个资源类型——通过增加来分配，通过减少来释放。与其他资源一样，它必须要封装起来。“字符串”对象就是这样一个封装器。事实上，因为引用计数是一个普通资源类型，所以，我们可能希望构建一个智能指针的等价体来处理引用计数对象。

```
template <class T>
class RefPtr
{
public:
  RefPtr(RefPtr<T> const & p)
  {
    _p = p._p;
    _p->IncRefCount();
```

```
  }
  ~RefPtr()
  {
    Release();
  }
  RefPtr<T> const & operator=(RefPtr<T> const & p)
  {
    if(this != &p)
    {
        Release();
        _p = p._p;
        _p->IncRefCount();
    }
    return *this;
  }
protected:
  RefPtr(T * p): _p(p){}
  void Release()
  {
    if(_p->DecRefCount()== 0)
        delete _p;
  }

  T * _p;
};
```

请注意，引用计数类型 T 至少必须提供两个方法：IncRefCount 和 DecRefCount。我们还假定类型 T 在传递给 RefPtr 的受保护构造函数前通过引用计数 1 来创建。

虽然它不是绝对必需的，我们可能希望类型 T 是一个实现了引用计数接口基类的后代。

```
class RefCounted
{
public:
  RefCounted(): _count(1){}
  int GetRefCount()const { return _count; }
  void IncRefCount()const { _count++; }
  int DecRefCount()const { return --_count; }
private:
  mutable int _count;
};
```

请注意一个有趣的细节：即使方法 IncRefCount 和 DecRefCount 修改了对象的数据，它们也被声明为 const。如果你将相关数据成员声明为 mutable，那你这样做编译器毫无异意。我们确实希望这些方法为 const（至少是其中的 IncRefCount），因为它们将在 RefPtr 内的 const 对象上被调用。两个拷贝构造函数及赋值运算符都将常量引用作为它们的参数，但是它们修改了它们的引用计数。所以，我们决定不考虑对象“修改”的引用计数更新，尽管严格来说理应这样做。当我们实现即写即复制（copy-on-write）时，它将发挥更大的作用。

为了演示的目的，让我们使用原来封装于类 StringRep（表示字符串）中的 StringVal 来创建一个引用计数字符串。通常，通过结合引用计数与字符串缓冲器，我们可以更有效地实现这一点。

```
class StringRep: public RefCounted
{
public:
  StringRep(char const * cstr)
    :_string(cstr)
  {}
  char const * c_str()const { return _string.c_str(); }
  void Upcase()
  {
    _string.Upcase();
  }
private:
  StringVal _string;
};
```

实际的字符串类是构建于 RefPtr 之上的。从内部上讲，它表示使用 StringRep 的字符串数据。

```
class StringRef: public RefPtr<StringRep>
{
public:
  StringRef(char const * cstr)
    : RefPtr<StringRep>(new StringRep(cstr))
  {}
  StringRef(StringRef const & str)
    : RefPtr<StringRep>(str)
  {}
  char const * c_str()const { return _p->c_str(); }
  void Upcase()
  {
    _p->Upcase();
  }
};
```

我们现在已经编写了这么多的类，如果不借助于图表（如图 11-2 所示），我们可能会遗漏其中一些类。

不同于 C-字符串的特殊构造函数，这里是没有数据复制。拷贝构造函数只是增加了字符串表示对象的引用计数。赋值运算符也是如此。所以，“复制”与按值传递 StringRef 开销相对少一些。应用这种实现方式只有一个小的问题。在对 StringRef 的其中一个拷贝调用 Upcase 后，其他所有拷贝都变成了大写。

```
StringRef strOriginal("text");
StringRef strCopy(strOriginal);
strCopy.Upcase();
```

```
// 原始者将变成大写!
std::cout << "The original: " << strOriginal.c_str()
          << std::endl;
```

这并非很常直观的行为。

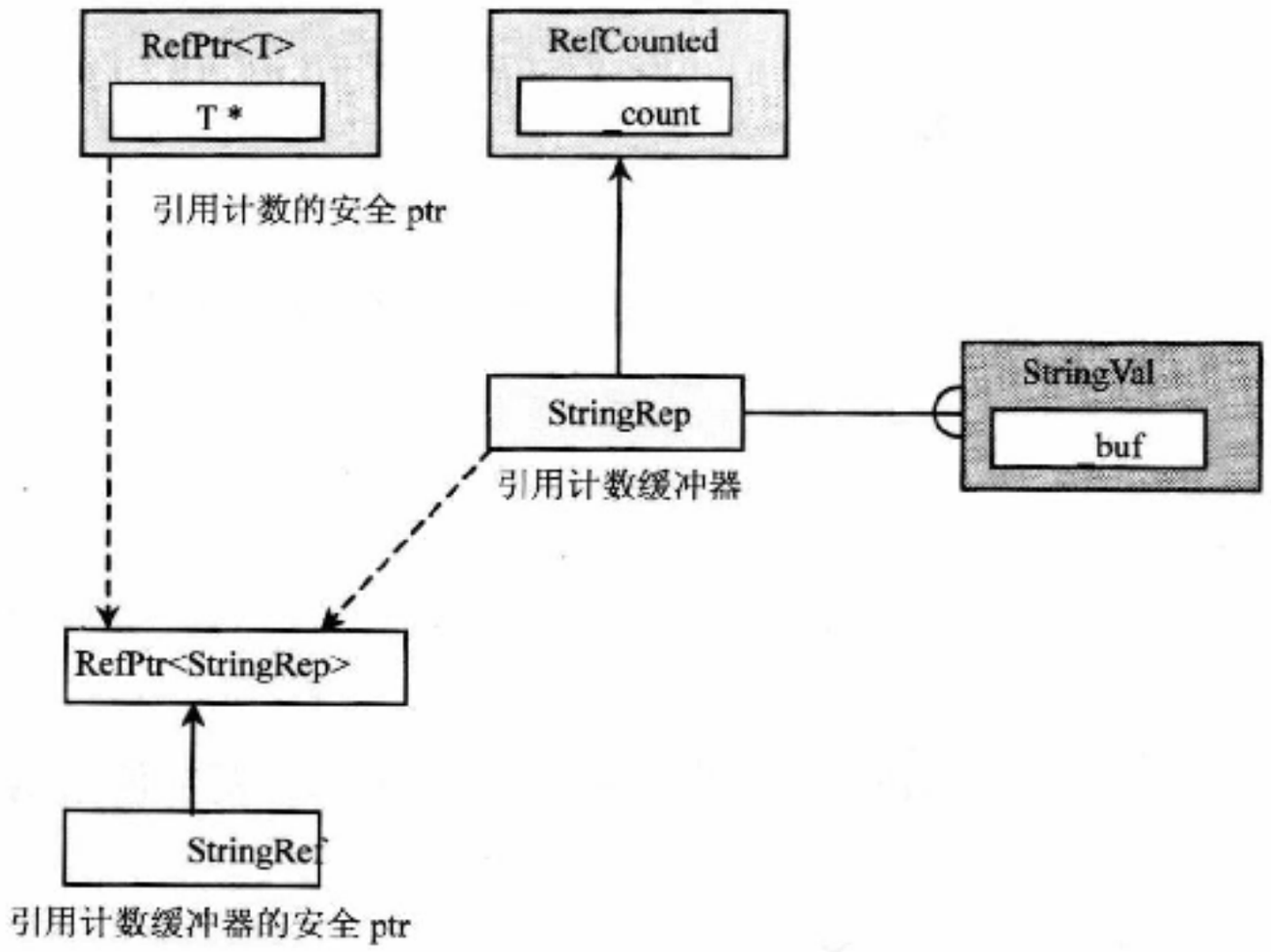


图 11-2　引用计数实现代码中的模板和类

string 3

不过，有一种办法可以两全其美。这就是即写即复制（copy-on-write），或简称为 COW。它的思想是共享多个拷贝间的单个表示，只要它们不需要进行任何修改。每个修改方法都必须确保它所修改的地方不是共享的。它检查它的表示的引用计数，如果计数大于 1，就进行私有复制。这样，复制可以尽量得以延迟。按值传递与共享表示的开销相同，但复制间的修改不再共享。

```
class StringCow: public RefPtr<StringRep>
{
public:
  StringCow(char const * cstr)
    : RefPtr<StringRep>(new StringRep(cstr))
  {}
  StringCow(StringCow const & str)
    : RefPtr<StringRep>(str)
  {}
  char const * c_str()const { return _p->c_str(); }
  void Upcase()
  {
    Cow();
```

```
    _p->Upcase();
  }
private:
  void Cow()
  {
    if(_p->GetRefCount()> 1)
    {
      // Clone it
      StringRep * rep = new StringRep(_p->c_str());
      Release();
      _p = rep;
    }
  }
};
```

从用户的角度看，这一实现法的巧妙之处在于，StringCow 的行为与 StringVal 完全一样，下至带有常量引用的拷贝构造函数及赋值运算符也是如此。除此之外，当它按值进行传递时，它的操作性能是最优的。

C++标准库也可以使用某些 COW 版本来实现字符串。

11.2 结束限制


Script 39

既然我们不再担心按值传递字符串，我们可以尝试改进更多的代码。例如，我们可以替换以下笨拙的代码块：

```
void Scanner::GetSymbolName(char * strOut, int lenBuf)
{
  assert(lenBuf > maxSymLen);
  assert(_lenSymbol < lenBuf);
  std::strncpy(strOut, &_buf [_iSymbol], _lenSymbol);
  strOut [_lenSymbol] = '\0';
}
```

当你使用字符串时，你不必担心发送合适大小的缓冲器。你让被调用者创建一个正确大小的字符串并按值返回它。下面是一种实现途径：

```
std::string Scanner::GetSymbolName()
{
  return std::string(&_buf [_iSymbol], _lenSymbol);
}
```

下面是接受方代码的样子：

```
std::string strSymbol = _scanner.GetSymbolName();
std::size_t id = _symTab.Find(strSymbol);
...
```

```
std::cerr << "Unknown function \"";
std::cerr << strSymbol << "\"\n";
```

注意，字符串 **strSymbol** 是一个限制了作用域的本地变量。它在流程控制离开此作用域时消失，而且释放它所占有的所有类型资源（或者至少要减小引用计数）。

你同时还能看到，字符串能发送到标准输出（或标准错误），而且像一个 C-字符串一样被打印。

实际上，事情变得更方便。你可以从标准输入直接读入文本到字符串。它的优点是，因为字符串可以自身动态调整大小，所以，实际上对接受文本的大小没有限制。用户可以任意输入想要输入的文本。那么，输入操作什么时候返回呢？必须找到一种终止字符中的办法。通常，标准输入一旦碰到空格字符及立即停止填充字符串。这就意味着，你可以按下面方式一次读入一个单词：

```
std::string str;
std::cin >> str;
```

我们程序的要求与之稍有不同。我们希望能一次读入一行。C++标准库有一个合适的函数来完成这一任务，所以，我们可以在 main 中直接使用它。

```
std::cerr << "> "; // 提示
std::string str;
std::getline(std::cin, str);
Scanner scanner(str.c_str());
```

小心 Bug! VC++ 6.0 提供的 C++标准库有一个漏洞：getline 需要用两行而不是一行作为它的终止符。访问 www.dinkumware.com 可以修正这一漏洞及其他错误。

让我们暂停一下。我们刚才取消了程序中的最后一个内在限制。再也没有对输入行长度或者标识符长度的限制了。文件 **params.h** 消失了。

这是为什么呢？我们开始使用动态数据结构。我们的程序会因此更复杂吗？不，不是的！事实上，它更简单了。还有其他一些原因可在程序中引入限制吗？几乎不可能！

11.3　去掉 C-字符串

Script 40

现在让我们从 main 到扫描器跟踪输入字符串。它在扫描器中保存一个纯粹 C-字符串作用不大。下面用一个标准字符串来改写：

```
class Scanner
{
public:
  Scanner(std::string const & buf);
  EToken  Token()const { return _token; }
```

```
EToken  Accept();
bool IsDone()const
{
  return _iLook == std::string::npos;
}
bool IsEmpty()const { return _buf.length()== 0; }
double Number()
{
  assert(_token == tNumber);
  return _number;
}
  std::string GetSymbolName();
private:
  void EatWhite();

 typedef std::string::size_type size_type;

  std::string const & _buf;
  size_type          _iLook;
  EToken             _token;
  double             _number;
  size_type          _lenSymbol;
  size_type          _iSymbol;
  static char        _whiteChars [];
};
```

标明 Scanner 到达了字符串结尾的自然办法是使用特殊索引值 string::npos。这一值由字符串的不同“查找”方法在到达字符串结尾时返回。例如，我们可以使用这样一个方法 find_first_not_of 来跳过缓冲器中的空格。

```
char Scanner::_whiteChars [] = " \t\n\r";
void Scanner::EatWhite()
{
  _iLook = _buf.find_first_not_of(_whiteChars, _iLook);
}
```

方法 find_first_not_of 带有两个参数，一个是要被忽略的以 null 终止的字符数组（在这里，数组包含空格、tab 符、换行符以及一个回车符），另一个是可选的起始索引，它的缺省值为 0。它返回忽略字符列表中不存在的第一个字符的出现索引。如果没有发现这样的字符，它返回 string::npos。

顺便提一下，只要比较同一种整数类型，值 string::npos 保证要大于任何有效索引。这是为什么我们确保使用同一种类型 size_type 作为字符串本身内部使用索引的原因。

这是 GetSymbolName 的另一种实现途径，它返回缓冲器的一个子字符串。

```
std::string Scanner::GetSymbolName()
{
  return _buf.substr(_iSymbol, _lenSymbol);
}
```

扫描器的其余实现代码几乎保持不变。我们只需要确保在 Accept 的末尾，如果缓冲器已经被占用了，我们就将位置设置成 string::npos。

```
if(_iLook == _buf.length())
    _iLook = std::string::npos;
```

而且像这样的代码，虽然不是最美观，但用于处理的字符串与用于直接的字符数组一样良好。

```
char * p;
_number = strtod(&_buf [_iLook], &p);
_iLook = p - &_buf [0];
```

在着手开发这一项目时，如果你决定使用 C++标准库的话，这一办法不是首选。但是，我想要给你演示对一个旧程序实施转换的可能性，并不需要一切从头开始。

11.4　探讨流

那么，使用 C++标准库解析输入的首选途径是什么呢？首先，我们不应该干扰从标准输入读入一行字符串。我们应该直接将流传递给扫描器。毕竟，我们不需要随机访问输入行——我们每次只是解析一个字符。除了在少数情况下，我们不必返回到字符串重新解析它（有一些语法需要如此，但在我们这里并不需要）。在少数必须返回的情况中，也只是通过一个字符。

Script 41

让我们从顶部开始。我们可以创建扫描器并作为一个参数将标准输入流传递给它。

```
cerr << "> ";  // 提示
Scanner scanner(std::cin);
```

下面是 Scanner 类的新定义。请注意，从 Scanner 的角度来看，cin 只是 std::istream 的一个例子（子类）。

```
class Scanner
{
public:
  explicit Scanner(std::istream & in);
  bool IsDone()const { return _token == tEnd; }
  bool IsEmpty()const { return _isEmpty; }
  EToken   Token()const { return _token; }
  void   Accept();
  std::string GetSymbolName()const { return _symbol; }
  double Number()const { return _number; }
private:
  void ReadChar();

  std::istream & _in;
```

```
  int             _look;    // 前视字符
  bool            _isEmpty;
  EToken          _token;
  double          _number;
  std::string     _symbol;
};
```

在这部分代码中，我没有做太多的重新组织。在_look 中我放置了一个前视字符。我还决定使用一个布尔标志_isEmpty 来显示构建扫描器时流为空的信息（一旦调用 Accept 后，我就不能再返过头来查看输入的开始）。我将 IsDone 的测试改为比较当前记号与 tEnd。最后，我需要一个字符串来记录从输入读取的最后一个符号变量的名字。

下面是 Scanner 的构造函数：

```
Scanner::Scanner(std::istream & in)
  : _in(in)
{
  Accept();
  _isEmpty =(Token()== tEnd);
}
```

Accept 方法需要少许重写。我们以前调用 EatWhite 的地方，现在我们调用 ReadChar。它像以前一样忽略空格，但它还初始化前视字符为第一个非空格字符。因为前视字符已经由 ReadChar 从输入流占用了，所以，我们在 Accept 中识别它后不必进行任何增加。

```
void Scanner::Accept()
{
  ReadChar();
  switch(_look)
  {
  case '+':
    _token = tPlus;
    // 无增加
  break;
  ...
  }
}
```

下面是 ReadChar 的实现代码：

```
void Scanner::ReadChar()
{
  _look = _in.get();
  while(_look == ' ' || _look == '\t')
    _look = _in.get();
}
```

我必须重新考虑输入结尾的处理。以前，我们使用 getline 来读取输入，实际上，我们从未必须处理一个换行符。通过定义，getline“吞食”了换行并终止了字符串（即 C-字符串版本添加一个 null；std::string 版本更新内部长度）。另一方面，get 方法照原样读取每一个字符，

包括换行符。因而，我让扫描器将换行符识别为输入的结尾。

```
case '\n':  // 输入结尾
case '\r':
case EOF:   // 文件结尾
_token = tEnd;
  break
```

偶尔，我会预先想到一些事，并决定让扫描器识别文件的结尾。当它碰到文件的结尾时，get 方法返回特殊的 EOF 值。这一值甚至不是一个字符（这就是定义 get 返回一个 int 而不是一个 char 的原因）。

标准输入流如何才能碰到文件的结尾呢？实际上有多种办法实现。首先，你可以从键盘输入它——在 DOS 中通过组合键 Ctrl+Z 输入。其次，程序可以通过重定向输入在命令行调用。你可以创建一个文本文件 calc.txt，填入计算器命令，再按如下方式调用它：

```
calc < calc.txt
```

操作系统将把文件内容插入程序的标准输入中并执行它。你将在标准输出上看到计算结果。除非你按下面方式重定向：

```
calc < calc.txt > results.txt
```

在后一种情况下，你只能看到标准错误（包括提示）的显示，计算结果将保存在文件 results.txt 中。

让我们继续改写扫描器。下面是识别一个数字时的代码：

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
  _token = tNumber;
  _in.putback(_look);
  _in >> _number; // Read the whole number
  break;
```

从标准输入读一个浮点数容易。当我们已经读取了数字的第一个字符——前视字符时，情况就变得复杂了。所以，在我们读取整个数字前，我们必须将前视字符放回流中。不要担心，这是一个简单的操作。输入流毕竟放入了缓冲器。当你调用 get 时，就从缓冲器中读取这一字符（除非缓冲器为空——此时，系统通过实际的输入读取进行了补充）。放回字符意味着将它放回到缓冲器中（通过 putback）。按这种方式实现输入流，它总是可以放回一个字符。

当读取一个标识符时所使用的技巧稍有不同。

```
default:
  if(std::isalpha(_look)|| _look == '_')
  {
    _token = tIdent;
    _symbol.erase(); // 擦除字符串内容
    do {
      _symbol += _look;
      _look = _in.get();
```

```
    } while(std::isalnum(_look));
    _in.putback(_look);
  }
  else
    _token = tError;
  break;
```

我们不必将前视字符放回读取标识符的开始。相反，我们必须放回最后一个字符，它不是标识符的一部分，所以下一个 ReadChar 调用可以看到它。

从字符串转换到流，我们没有忽略一些通性吗？尽管你可以总是从流转换到字符串（例如使用 getline）。反之可行吗？不要担心！转换字符串到流同样容易。对应的类就是 istringstream，它在头<sstream>中定义。因为 istringstream 继承于 istream，所以扫描器不会发现它的不同之处。例如，我们可以：

```
std::istringstream in("sin(2 * pi / 3)");
Scanner scanner(in);
```



Script 42

我们已经揭开了标准库的面纱，而且我们已经找到了许多有用的资源。它实在值得我们去学习，你就不用再从零开始实现你的解决方案了。

第12章 持久性

大部分程序都具备在不同过程间持久保存数据的机制。数据通常保存在磁盘文件中，但也可以选择其他途径（一个数据库、Windows 注册表、CD-ROM 等等）。在磁盘上构建数据也有不同的途径。一些程序使用文本文件，其他程序则选择二进制格式。

文本文件易于处理、调试或在需要时进行修正。缺点是用户使用文本编辑器时也容易导致它的崩溃。二进制文件的内容不可直接人工识别，它在这方面更安全，而且它通常占用空间少。但是当用一个二进制文件保存程序的状态时，你必须提供一个设计可靠的二进制格式。这是面向对象方式带来的好处。

作为程序修改的一个练习，在准备添加持久保存数据机制前，我们将首先简化计算器的顶层结构。

12.1 Calculator 对象

Script 43

考察 main：其中的对象太多——符号表、函数表及 store。所有三个对象生存期相同——都为程序的执行期。它们必须按特殊的顺序初始化，而且均被传递给解析器的构造函数。能将它们组合到一个对象吗？将这一对象命名为 Calculator 类。按正确顺序将它们嵌入这一个类中与初始化正确顺序有关。

```
class Calculator
{
   friend class Parser;
 public:
   Calculator()
     : _funTab(_symTab),
```

```
      _store(_symTab)
  { }
private:
  Store & GetStore(){ return _store; }
  PtrFun GetFun(std::size_t id)const
  {
    return _funTab.GetFun(id);
  }
  bool IsFunction(std::size_t id)const
  {
    return id < _funTab.Size();
  }
  std::size_t AddSymbol(std::string const & str)
  {
    return _symTab.ForceAdd(str);
  }
  std::size_t FindSymbol(std::string const & str)const
  {
    return _symTab.Find(str);
  }

  SymbolTable    _symTab;
  Function::Table _funTab;
   Store          _store;
 };
```

当然，我们现在必须在 main 和 parser 中作出适当的更改（应该是简化）。下面是 parser 声明及其实现中的几个例子：

```
class Parser
{
public:
  Parser(Scanner & scanner, Calculator & calc);
  ...
private:
  ...
  Scanner          & _scanner;
  std::auto_ptr<Node> _pTree;
  Status           _status;
  Calculator       & _calc;
};
// Factor := Ident
if(id == SymbolTable::idNotFound)
{
  id = _calc.AddSymbol(strSymbol);
}
pNode = std::auto_ptr<Node>
        (new VarNode(id, _calc.GetStore()));
```

你注意到变化之处了吗？我只是在项目的顶层作了另一个主要修改。事实上，这样做几乎微不足道，它对编译器的帮助很少。让我一步一步描述其过程。

从 main 中定义（构建）符号表、函数表以及定义 store 的地方开始。将它们用新对象 calculator 替换，声明类为 Calculator，并且为它编写一个构造函数。现在，如果你不知道下一步该如何做，请启动编译器。编译器将立即告诉你下一步该做什么：必须修改 parser 的构造函数。你必须将它传递给计算器而不是它的三个独立部分。此时，你可能注意到更改 Parser 的类声明让它保存一个指向 Calculator 引用的必要性。或者，你可以再次运行编译器，让编译器来提醒你怎么做。

接下来，你会看到实现 Parser 时的所有编译器错误。你可以逐个修正它们，在需要的时候给 Calculator 添加新方法。整个过程非常简单，即使是一个刚刚着手此项目的实习生也能轻松完成。



Script 44

既然我们现在拥有了头文件 Calc.h，我们也应该利用这次机会将 main 移出文件 Calc.cpp。应该将 main 放在哪儿呢？为此目的创建一个单独的文件 Main.cpp 如何？当有新同事加入到开发团体中时，他或她将立即知道从哪里开始阅读你的代码。

修改工作给我们的启示是：项目高层结构的改进永无止境。第一次时就完美无缺几乎是不可能的。

顺便提醒读者，你刚才见到的是就是自顶向下程序修改方法。从顶部开始，编译器引导你逐渐进入实质性的实现细节。这是自顶向下方法学的第三部分，它包括：

- 自顶向下设计
- 自顶向下实现
- 自顶向下修改

无论如何强调自顶向下方法学的重要性也不过份。我至今还未看到一个清晰、优秀的自底向上设计代码块。你可能会听到别人说，自顶向下设计在某些方面做得好，而自底向上在另外一些方面做得好。也有人会说，先从中间着手再向两边扩展是最好的办法。不要轻信他们的话!

事实是，当你缺少整体规划时，自底向上开发模式更自然——你的目标不是编写一个特定的程序，而是即兴发挥而已。例如，这是一种学习可能用到的晦涩子系统接口的容易途径。如果你不是十分擅长于设计，或者你不喜欢殚精竭虑地构思，只愿意动手编写实际的代码，那么自底向上也是合适的模式。如果你乐于长时间地调试或有人替你效劳（但愿不是最终用户），这绝对是上上之选。

如果你采用自底向上的思想，你将最终不得不重新设计，而且永远也不能开发出有专业水准的代码。在行家眼里，你的程序像那些电子工程一样，在电路板上用无线电组件组成，杂乱的电线到处都是，而且电池也用橡皮带绑在一起。

我最终决定去掉顶层混乱并引入 Calculator 的真正原因是简化添加新功能块的工作。每次主管人员要求你添加新的功能，抓住这次机会通过少量修改现有代码就可以出色完成任务。代码至少要改写三次，否则得到的代码不是非常完美。我不是在开玩笑！

改写的意思不是摒弃一切从头开始。只是偶而花一些时间来改进项目各部分结构。得到的回报将非常丰厚，它能缩短开发周期。那当然，如果你的上司盲目自大，你将很难让他信这一套做法。他们总是在不停地叫喊一些废话，如“如果它没有崩溃，就不要去修改它”或“如果我们明天不发布，我们都死定了”。此时你别无选择只有放弃!你永远也不能按正确方式行事，你将花费更多更多的时间来更正这些低质量程序中的五花八门的 Bug。艰难的维护工作在所难免了！

言归正传。在项目结束之时，假想上司告诉我们：如果我们没有提供保存和恢复计算器状态的命令，我们就死定了。幸运的是，我们可以轻易地在程序中添加此功能，同时完成更多的清理工作。

12.2 命令解析器

下面讨论按一种有序方式添加新的功能。我们必须为用户提供一种输入命令的方式。到目前为止，我们已经可以输入退出命令了——将一个空行解释成“退出”。现在，我们想要另外添加两个命令——“保存”和“恢复”，我们最好找到一种更通用的解决方案。我可以不必讲授这些内容，但是……

> 无论什么时候，只要多于两种特殊情况，你就应该把它们一般化。

Script 45

当前，计算器接受用户输入的表达式。让我们在命令前面加上一个惊叹号来区分命令与表达式。惊叹号有命令某人做某事的自然内涵。我们将它用作一个前缀而不是用作后缀来简化解析工作。我们还将把退出命令（quit）写成!q。我们甚至会在计算器启动时提醒用户。

```
std::cerr << "\n!q to quit\n";
```

新 Scanner 方法 IsCommand 将检查前导惊叹号（我们可以给 Scanner 添加一个新的记号tCommand）。一旦我们确定了某行文本是一个命令，我们就创建了一个简单的 CommandParser 来解析和执行它。

```
if(!scanner.IsEmpty())
{
  if(scanner.IsCommand())
  {
    CommandParser parser(scanner, calc);
    status = parser.Execute();
  }
```

```
    else
    {
      Parser  parser(scanner, calc);
      status = parser.Parse();
      if(status == stOk)
      {
        double result = parser.Calculate();
        std::cout << result << std::endl;
      }
      else
      {
        std::cerr << "Syntax error\n";
      }
    }
}
```

下面是新类 CommandParser:

```
class CommandParser
{
  enum Ecommand
  {
    comSave,
    comLoad,
    comQuit,
    comError
  };
public:
  CommandParser(Scanner & scanner, Calculator & calc);
  Status Execute();
private:
  Status Load(std::string const & fileName);
  Status Save(std::string const & fileName);
  Scanner &    _scanner;
  Calculator & _calc;
  ECommand     _command;
};
```

解析命令的方式:

```
CommandParser::CommandParser(Scanner & scanner,
                             Calculator & calc)
  : _scanner(scanner), _calc(calc)
{
  assert(_scanner.IsCommand());
  _scanner.Accept();
  std::string name = _scanner.GetSymbolName();
  switch(name [0])
  {
    case 'q':
```

```
    case 'Q':
      _command = comQuit;
      break;
    case 's':
    case 'S':
      _command = comSave;
      break;
    case 'l':
    case 'L':
      _command = comLoad;
      break;
    default:
      _command = comError;
      break;
  }
}
```

请注意，我们重用 Scanner 方法 GetSymbolName 来读取命令字符串。

在 name[0]中，我们使用数组索引访问字符串的第一个字符。你可以按索引一个 C-字符串的同样方式索引一个标准字符串——它定义了自己的 operator []。load 和 save 命令需要一个参数，即文件名。我们再次使用方法 GetSymbolName 从扫描器读取它。

```
Status CommandParser::Execute()
{
  _scanner.AcceptCommand();
  std::string nameFile;
  switch(_command)
  {
  case comSave:
    nameFile = _scanner.GetSymbolName();
    return Save(nameFile);
  case comLoad:
    nameFile = _scanner.GetSymbolName();
    return Load(nameFile);
  case comQuit:
    std::cerr << "Good-bye!" << std::endl;
    return stQuit;
  case comError:
    std::cerr << "Error" << std::endl;
    return stError;
  }
  return stOk;
}
```

我们使用新方法 AcceptCommand 来接受命令并读取后面的字符串。字符串（可能是一个文件名）必须由一个空格终止。注意，我们不能使用 Scanner 的常规方法 Accept，因为它只读入 C++标识符形式的字符串。例如，当它读到一个句点时就会停止，句点是一个文件名

中的完全有效组成部分（如果我们更严格一些，我们甚至会对文件名中的空格也作出规定。尽管如此，我们也可以要求它们放在一个引号中）。

```
void Scanner::AcceptCommand()
{
  ReadChar();
  _symbol.erase();
  while(!std::isspace(_look))
  {
    _symbol += _look;
    _look = _in.get();
  }
}
```

按惯例，我们应该为 Load 和 Save 命令提供简单的代码片段，并在进一步处理前测试我们的程序。

12.3　串行化与反串行化

我们常常将数据结构想像成两维、甚至三维的动物（譬如解析树、哈希表或多维数组）。另一方面，磁盘文件为一维结构——它是线性的。当你写入文件时，就是按顺序一个内容接一个内容地写入。因此将它称做串行化（serialization）。保存数据结构意味着将多维思想转换成一维表达形式。当然，计算机内存实际也是一维的。内存中的数据结构也按某种方式串行化。有一些数据，例如多维数组，它们通过编译器串行化；其他数据结构通过指针使之适应于线性内存。遗憾的是，指针在运行程序当前实例之外的其他场合无意义。指针不能保存为文件、关闭程序、重新启动程序、读取文件以及让新读取指针指向以前的同一个数据结构。

为了串行化一个数据结构，必须提供一个功能完备的过程来处理它，即依次访问它的每个元素。例如，你可以通过跟踪下一个指针遍历一个简单链表，直到到达列表的结尾为止。如果为循环列表，你必须记住初始指针，而且每走一步，都要将它与下一个指针进行比较。一个二维树可以通过首先遍历左边子代然后遍历右边子代来完成遍历（注意这里的递归性描述）。对每一个数据结构，至少有一个确定性的过程来遍历它，但是这一过程的复杂性可能有大有小。

一旦你知道如何遍历一个数据结构后，你就知道如何串行它了。你知道了如何来访问这一结构中的每个元素，按顺序进行——扫描它的一种连续方式。在每个数据结构的底层，你总可发现简单的内置类型，如 int、char、long 等，它们可以按一套简单规则写进文件中——稍后我们再来讨论这一点。如果你知道如何串行每个基本元素，你就达到了目的。

只有我们知道如何恢复数据结构（将它从文件反串行化到内存）时，对它的串行化才有意义。知道原始的串行化过程有助于我们按照相同的步骤反串行化，此时，我们才读文件并写入内存，而不是串行化写入文件。不过，我们必须确保这一过程准确无误。例如，我们必须知道何时停止读一个给定数据结构的元素。我们必须知道数据结构在哪里结束。串行化过程中的线

索不会出现在磁盘上。例如，一个链表用一个空指针在最后一个元素处作为下一个指针。但是，如果我们决定不保存指针，我们怎样才能知道何时到达列表的结尾呢？当然，我们可以决定保存指针，提供何时停止的线索。甚至更好的是，我们可以在列表前面保存元素计数。

在我们反串行它们之前，由于需要知道数据结构的大小，所以对串行化的顺序提出了额外的限制。当我们串行化程序的一部分数据时，其他部分保存在内存中。我们经常可以通过查看其他数据结构推断给定数据结构的大小。反串行化时就没有这么轻松了。我们必须确保这些数据首先被反串行化，或者给串行化映像添加一些冗余，如多次保存计数。例如，一个类包含一个指向某动态分配数组及当前数组大小的指针，这就是一个典型的例子。对于指针或计数，先提供哪一个成员没有影响。但是，当串行化这一对象时，我们必须首先保存计数，然后再保存数组内容。否则，我们不能分配适当数量的内存或读取正确的条目数量。

保存多态数据结构时可能会导致另一种二义性。例如，一个二元节点包含两个指向 Node 的指针。我们串行化它时不会发生问题——它可以通知两个子代通过调用适当的虚函数来串行化它们。但是在反串行化这一节点时，我们怎样才知道每个子代的真正类型呢？我们必须在开始反串行化之前知道这一点。这是为什么串行化多态数据结构的映像必须从一些确定数据结构的类的代码开始的原因。基于以上代码，反串行化器才能够调用对应的构造函数。

让我们现在回到计算器项目中并实现 Calculator 的数据结构的串行化和反串行化。首先，我们必须创建一个输出文件。我们将在一个串行流对象中封装一个文件。串行流可以接受基本数据类型的数字（例如，long、short 和 double）以及一些简单聚集（如字符串）并将它们写入文件。

注意，我没有提到最常见的类型——整数类型（integer）。这是因为整数的大小与系统有关。假设你串行化一个包含整数的数据结构并将它发送到磁盘，或者通过 e-mail 发送给其他人，而此人在不同的处理器上安装有同一个程序的某个版本。你的程序可能按 2 个字节写入整数，但它们的程序可能要求是 4 个字节甚至是 8 个字节的整数。这就是为什么在串行化时，我们通常将整数这样的与系统有关的类型转换成与系统无关的类型，如 long。事实上，不仅大小有所区分，字节的顺序也同样重要。

实质上有两种处理器：一种使用 big-endian[1]，另一种使用 little-endian 顺序（有些处理器可以使用其中任何一种）。例如，short 或 long 可以按“重要者优先”（most-significant-byte-first）保存，也可以按“次要者优先”（least-significant-byte-first）方式保存。Intel 处理器家族按“次要者优先”（little-endian 方式）保存，而 Motorola 处理器家族与之相反。所以，如果你希望你的程序在 Wintel（运行在 Intel 处理器上的 Windows）与 Macintosh 间协作，那么在串行化简单数据类型时必须考虑字节的顺序。当然，如果你不准备在两大阵营中同时发布你的程序，你可以安全地忽略这一点。

[1] 译者注：依照 CPU 设计的不同，当资料宽度超过一个位元组（byte）以上时，将高位元组摆在前面称为 big-endian，相反的方式称为 little-endian。

尽管如此，我仍然相信，在大部分情况下，你应该留意变量大小类型，并将整数或枚举转换成定长类型。

一旦为程序提供了一种磁盘格式后，这种格式就永远不再改变，这是一种理想的情况。但实际上，这种想法非常天真。你可以改变格式，至少可以让程序拒绝装载不能识别的格式。

总是将版本号保存为磁盘数据结构的一部分。

Script 46

为了实现串行化，我要做的是创建一个流，将版本号写入，并且通知计算器来串行化它。顺便提醒你，我们现在可以体会前面将几种对象组合到 Calculator 对象中的好处了。

```
const long Version = 1;
Status CommandParser::Save(std::string const & nameFile)
{
  std::cerr << "Save to: \"" << nameFile << "\"\n";
  Status status = stOk;
  try
  {
    Serializer out(nameFile);
    out.PutLong(Version);
    _calc.Serialize(out);
  }
  catch(char const * msg)
  {
    std::cerr << "Error: Save failed: "
              << msg << std::endl;
    status = stError;
  }
  catch(...)
  {
    std::cerr << "Error: Save failed\n";
    status = stError;
  }
  return status;
}
```

当反串行化时，我们遵循相同的步骤，除了现在执行的是读而不是写、反串行化而不是串行化。而且，如果版本号不匹配，我们就拒绝装载。

```
Status CommandParser::Load(std::string const & nameFile)
{
  std::cerr << "Load from: \"" << nameFile << "\"\n";
  Status status = stOk;
  try
  {
    DeSerializer in(nameFile);
```

```
    long ver = in.GetLong();
    if(ver != Version)
        throw "Version number mismatch";
    _calc.DeSerialize(in);
  }
  catch(char const * msg)
  {
    std::cerr << "Error: Load failed:"
              << msg << std::endl;
    status = stError;
  }
  catch(...)
  {
    std::cerr << "Error: Load failed\n";
    // 数据结构可能崩溃
    throw;
  }
  return status;
}
```

Script 47

在 Calculator 中有两个对象我们要保存到磁盘上——符号表以及 store——，个保存变量的名字，另一个保存它们的值。下面是对应的代码：

```
void Calculator::Serialize(Serializer & out)const
{
  _symTab.Serialize(out);
  _store.Serialize(out);
}
void Calculator::DeSerialize(DeSerializer & in)
{
  _symTab.DeSerialize(in);
  _store.DeSerialize(in);
}
```

Script 48

符号表包含一个映射字符串到整数的字典，还有一个包含当前 ID 的变量，而且这也是遍历符号表的最简单方式。为了遍历标准映射，我们将使用它的迭代器。首先，我们必须保存元素的计数，我们根据它可以知道反串行化时读入了多少元素，然后，我们将迭代访问整个映射并保存（字符串、id）对。请注意，std::map 的迭代器指向 std::pair，它拥有第一个和第二个数据成员。根据前面的讨论，我们将 ID 保存为一个 long。

```
void SymbolTable::Serialize(Serializer & out)const
{
  out.PutLong(_dictionary.size());
```

```
std::map<std::string, std::size_t>::const_iterator it;
for(it = _dictionary.begin();
    it != _dictionary.end(); ++it)
{
  out.PutString(it->first);
  out.PutLong(it->second);
}
out.PutLong(_id);
}
```

反串行化器必须按数据串行化的顺序读入：首先是字典，然后是当前 ID。当反串行化映射时，我们首先读入它的大小。然后，我们读取字符串和 long 对并添加到映射中。我们在这里将映射当做一个关联数组。请注意，我们首先清除已有字典——必须这样做，否则，它可能会与对应于不同字符串的相同 ID 冲突。

```
void SymbolTable::DeSerialize(DeSerializer & in)
{
  _dictionary.clear();
  int len = in.GetLong();
  for(int i = 0; i < len; ++i)
  {
    std::string str = in.GetString();
    std::size_t id = in.GetLong();
    _dictionary [str] = id;
  }
  _id = in.GetLong();
}
```

注意，对每一个串行化过程，我们要立即写下它的对应过程——反串行化过程。这样才能保证两者相匹配。

Script 49

store 的串行化也非常简单：首先是大小，然后是（double、bool）对序列。

```
void Store::Serialize(Serializer & out)const
{
  std::size_t len = _cell.size();
  out.PutLong(len);
  for(std::size_t i = 0; i < len; ++i)
  {
    out.PutDouble(_cell [i]);
    out.PutBool(_isInit [i]);
  }
}
```

当我们反串行化 store 时，我们首先清除以前的值，读取大小，然后逐个读取（double、bool）对。用新值填充两个矢量时有几种选择。一种选择是将它们逐个推回。因为我们预先知道了条目的数量，我们可以调用方法 reserve 在矢量中保留空间。在这里，我决定重新调整

矢量的大小，然后将它看作数组。调整大小时，用 0 填充 double 矢量，用 false 填充 bool 矢量（0 和 false 是这些类型的缺省值）。

在 reserve 和 resize 间有一个重要的区别。大部分标准容器有其中一个方法，或者两个均有。reserve 确保添加元素当保留容量时不重新分配，例如，使用 push_back 来完成。当我们预先知道所需容量时，这是一种好的优化措施。对于一个矢量，不存在重新分配也意味着迭代器、指针或对矢量元素的引用不会突然由于内部重新分配而无效。

不过，reserve 没有改变容器的大小；但 resize 改变了容器的大小。当你调整容器的大小时，就添加了新元素（因而，如果容器保存的对象没有缺省构造函数或缺省值，你就不能改变容器的大小。）

- reserve——改变容量但不改变大小。
- resize——改变大小。

你可以通过调用 capacity 询问当前容器的容量。当然，调用 size 可以得知它的大小。

```
void Store::DeSerialize(DeSerializer & in)
{
  _cell.clear();
  _isInit.clear();
  std::size_t len = in.GetLong();
  _cell.resize(len);
  _isInit.resize(len);
  for(std::size_t i = 0; i < len; ++i)
  {
    _cell [i] = in.GetDouble();
    _isInit [i] = in.GetBool();
  }
}
```

Script 50

最后，让我们看看反串行化器流的实现代码。它是输出流顶部的极薄层。

```
#include <fstream>
using std::ios_base;

const long TruePattern = 0xfab1fab2;
const long FalsePattern = 0xbad1bad2;

class DeSerializer
{
public:
  DeSerializer(std::string const & nameFile)
    : _stream(nameFile.c_str(),
              ios_base::in | ios_base::binary)
```

```
{
  if(!_stream.is_open())
      throw "couldn't open file";
}
long GetLong()
{
  if(_stream.eof())
      throw "unexpected end of file";
  long l;
  _stream.read(reinterpret_cast<char *>(&l),
              sizeof(long));
  if(_stream.bad())
     throw "file read failed";
  return l;
}
double GetDouble()
{
  double d;
  if(_stream.eof())
     throw "unexpected end of file";
  _stream.read(reinterpret_cast<char *>(&d),
              sizeof(double));
  if(_stream.bad())
     throw "file read failed";
  return d;
}
std::string GetString()
{
  long len = GetLong();
  std::string str;
  str.resize(len);
  _stream.read(&str [0], len);
  if(_stream. bad())
     throw "file read failed";
  return str;
}
bool GetBool()
{
  long b = GetLong();
  if(_stream.bad())
     throw "file read failed";
  if(b == TruePattern)
     return true;
  else if(b == FalsePattern)
     return false;
  else
```

```
        throw "data corruption";
      return false; // 避免编译器警告
    }
private:
  std::ifstream _stream;
};
```

在这里发生了几件有趣的事。首先，我们传递给 ifstream::open 的那些奇怪标记是什么？第一个标记 ios_base::in 的意思是，我们正打开用于输入的文件。第二个标记 ios_base::binary 告诉操作系统，我们不需要任何回车换行翻译。

回车换行的意思是什么？这是 DOS 文件系统的最大失误之一，不幸的是，所有的 Windows 家族都继承了这一失误。DOS 的发明者认为系统必须将单个字符'\n'转换成'\r'、'\n'对。理由是，当你打印一个文件时，打印机将回车符'\r'解释成回到当前行正式起始处的命令，将换行'\n'解释成移到下一行的命令（不一定移到起始处）。所以，需要使用两个字符让打印机移到下一行的起始处。现在，当我们使用能理解 PostScript 的激光打印机打印 WYSIWYG[1] 文档时，这一思想似乎太怪异了，对于旧操作系统 UNIX，情况更是如此，它处理这类问题时没有提供低级文件系统服务。

无论如何，如果你的愿望是在一个文件中保存数据字节，你必须记住按“二进制”方式打开它，否则会产生意想不到的结果。顺便说一下，缺省方式为 ios_base::text，它就执行着这一不适宜的字符转换。

另一个有趣的地方是，方法 ifstream::read 读取数据到字符缓冲器中——它要求 char *作为它的第一个参数。当我们想读入一个 long 时，我们不能只是给它传递 long 的地址——编译器不知道如何将一个 long *转换成一个 char *。在这种情况下，我们必须强制让编译器相信我们。我们希望将 long 分解成子字节（这里忽略了 big-endian/little-endian 问题）。使用 reinterpret_cast 是一个合理的途径。我们实质上是告诉编译器将 long 占据的内存块“重新解释”成字符系列。应用运算符 sizeof，我们可以知道一个 long 包含多少个字符。

现在应该向读者解释强制类型转换（cast）的不同类型了（参见表 12-1）。还有一个 dynamic_cast，我们在这里不作讨论。

表 12-1　　强制类型转换

使用此类型转换	转 换 结 果
const_cast	去掉常量属性
static_cast	转换相关类型
reinterpret_cast	转换指向无关类型的指针（引用）

[1] 译者注：WYSIWYG 为“What You See Is What You Get”的缩写，意思是“所见即所得”。

下面是一个 const_cast 例子：

```
char const * str = "No modify!";
char * tmp = const_cast<char *>(str);
tmp [0] = 'D';
```

为了理解 static_cast，可以将它看作隐式转换的反过程。无论什么时候将类型 T 隐式转换成类型 U（也就是说，T“是一个”U），就可以使用 static_cast 来执行相反的转换。例如，字符可以隐式转换成 int：

```
char c = '\n';
int i = c;       // 隐式转换
```

所以，当你需要把 int 转换成 char 时，必须使用 static_cast：

```
int i = 0x0d;
char c = static_cast<char>(i);
```

要不然，如果你有两个类，Base 和 Derived: public Base，你可以隐式将 Derived 的指针转换成 Base 的指针（Derived“是一个”Base）。因而，你可以使用 static_cast 进行相反的转换：

```
Base * bp = new Derived; // 隐式转换
Derived * = static_cast<Base *>(bp);
```

你必须意识到强制类型转换的危险性，使用时必须小心翼翼。使用强制类型转换后，错误检查的工作由程序员负责，而不再由编译器负责了。要尽量避免使用强制类型转换。串行化及反串行化是使用强制类型转换的特例，因为它们需要低级操纵类型。

最后要注意我们保存布尔值的奇怪方式。一个布尔值实际上只需要一个位的存储空间。但是，因为我们不希望分解字节（甚至是 long），所以在这里使用了冗余。在原理上讲，我们可以将值 true 保存为 1，将 false 保存为 0。但是，写入一个 0 与写入一个任意值的开销一样大。区别在于，在文件中用 0 比用 0xbad1bad2 这样的值更普遍一点。所以，当我再读值 0xbad1bad2 而且需要一个布尔值时，我就确信自己读到的是一个有意义的数据，而不是一些无用的垃圾。这只是利用冗余进行一致性检查的其中一种途径。

输出串行化流为 DeSerializer 的镜像。

```
class Serializer
{
public:
Serializer(std::string const & nameFile)
  : _stream(nameFile.c_str(),
            ios_base::out | ios_base::binary)
{
  if(!_stream.is_open())
       throw "couldn't open file";
}
void PutLong(long x)
{
  _stream.write(reinterpret_cast<char *>(&x),
```

```
                sizeof(long));
  if(_stream.bad())
      throw "file write failed";
}
void PutDouble(double x)
{
  _stream.write(reinterpret_cast<char *>(&x),
                sizeof(double));
  if(_stream.bad())
      throw "file write failed";
}
void PutString(std::string const & str)
{
  std::size_t len = str.length();
  PutLong(len);
  _stream.write(str.data(), len);
  if(_stream.bad())
      throw "file write failed";
}
void PutBool(bool x)
{
  long l = x? TruePattern: FalsePattern;
  PutLong(l);
  if(_stream.bad())
      throw "file write failed";
 }
private:
  std::ofstream _stream;
};
```

组合赋值与条件有一种简写法，如下所示：

```
long l = x? TruePattern: FalsePattern;
```

它等价于：

```
long l;
if (x)
  l = TruePattern;
else
  l = FalsePattern;
```

三元（意思是有三个参数）运算符 **A? B: C** 首先求 A 的值。如果 A 为真，则求 B 的值并返回，否则就求 C 的值并返回。有一点要说明：与 C 不一样，在 C++中，三元运算符返回一个左值，所以，它可用于赋值的左边。但我并不是向你推荐这种样式！

在 C++中，令人更加晦涩难懂的运算符是逗号序列运算符（我们在以前碰到过它）。表达式

```
A, B
```

首先求 A 的值，然后求 B 的值并返回。A 的求值只是整个操作的副作用。逗号运算符最常用的场合是组合两个表达式，而最终只需要其中的一个表达式，如下面的双重循环：

```
for(int i = 0, j = 0; i < maxI && j < maxJ; ++i, ++j)
```

顺便说一下，第一个逗号分隔两个相同类型变量的声明（通过初始化完成）。++i 和 ++j 间的第二个逗号为序列运算符。

注意，我们读取或写入一个文件时必须十分小心。因为程序不能完全控制磁盘，如果磁盘空间不够，写入操作就可能失败。任何时候都可能发生这种情况，因为我们不是文件系统的唯一客户——还可能有其他应用程序和系统服务要使用磁盘空间（当然也可能同时释放一些空间）。

读取文件更需小心，因为我们甚至不能确定文件中即将读到的内容。如果硬件发生故障（磁盘扇区损坏），读就可能失败，而且我们必须应付各种各样的不利因素。其他应用程序可能正在使用这一文件，而且可能正在删减、编辑或覆写此文件。我们甚至不能确定我们即将解析的文件是否已经由程序创建。用户可能给程序错误地或恶意地传递某些可执行文件、电子表的名字或 autoexec.bat。

我们已经对这些错误或崩溃筑起了第一道防线——这就是版本号。Command Parser 从文件中读取的头 4 个字节必须与当前版本号匹配，否则，它将拒绝装载它。不过，此时显示的错误信息“Version number mismatch”（版本号不匹配）有一些误导。更好解决方案是分出若干额外的字节，然后在文件中添加一个幻数。有些人使用自己首创的幻数，他们希望将来有一天，他们可以告诉其后代：“你看到每个文件开头这种类型的几个字节了吗？这就是你们的妈妈（爸爸、祖母或祖父）的首创”。当然，前提是这一应用程序或系统能够经受足够时间的考验，而且没有成为反面软件工程范例。

12.4　内存中的串行化及反串行化

数据结构的串行化并非必须与文件的保存相关联。有时，你只是想在内存块中保存一些数据结构，尤其是你想将它传递给另一个应用程序的时候。程序可以相互对话，通过共享内存或其他渠道传递数据（此时我们会想起 Windows 的剪切板）。你也可能想在网络间按包的形式发送。在这些情况下，你不能仅仅传递嵌入数据中的指针。你必须改变数据的格式。

不管输出到一个文件还是到内存，串行化过程都是相同的。事实上，如果你的数据结构可串行化（它具备 Serialize 和 DeSerialize 方法），你所需要的是修改 Serializer 和 DeSerializer 的实现代码在内存中串行化它。更好的是，你可能让这些类抽象化——将方法 PutLong、PutDouble、PutBool 以及 PutString 变成纯虚方法，并且提供两个不同的实现过程，一个写入文件，另一个写入内存。当然，你应该对反串行器进行相同的修改。

在文件和内存块间有一个大的区别——文件随着写入内容的增加而变大，但内存块的大

小固定。有两个选择——你既可以按需要增大缓冲器，也可以预先计算内存大小并预分配整个缓冲器。事实表明，计算可串行化数据结构的大小要容易得多。我们所需要的是实现另一个 Serializer 接口，即计数串行化器。计数串行化器不写入任何内容，它只是合计要求写入的所有不同数据类型。

```
class CountingSerializer: public Serializer
{
public:
  CountingSerializer()
  : _size(0){}
std::size_t GetSize()const { return _size; }
void PutLong(long l)
{
  _size += sizeof(long);
}
void PutDouble(double d)
{
  _size += sizeof(double);
}
void PutString(std::string const & str)
{
  _size += sizeof(long); // 计数
  _size += str.length();
}
void PutBool(bool b)
{
  _size += sizeof(long);
}
private:
  std::size_t _size;
};
```

例如，如果你想计算串行化计算器所需的文件或内存缓冲器大小，你可以调用带有一个计数串行化器的 Serialize 方法。

```
CountingSerializer counter;
calc.Serialize(counter);
std::size_t size = counter.GetSize();
```

请记住：为了完成这一任务，Serializer 的所有方法必须声明为 virtual 。

12.5 多重继承

Script 51

为了使一个类可串行化，你必须添加两个方法：Serialize 和 DeSerialize，并编写相应的

实现代码。然后，再创建一个单独的抽象类（一个纯接口）来抽象化它的行为。

```
class Serializable
{
public:
  virtual void Serialize(Serializer & out)const = 0;
  virtual void DeSerialize(DeSerializer & in)= 0;
};
```

所有可串行化类都应该继承于 Serializable 接口。

```
class Calculator: public Serializable
class SymbolTable: public Serializable
class Store: public Serializable
```

这样做的优点是什么？即使你继承于 Serializable，你仍然必须在类中添加这两个方法的声明，而且提供它们的实现代码。

假如一个新的程序员加入到你的开发团体，而且他或她必须为此项目添加一个新类。有一天，此人给你发过来一封 e-mail 并询问："我该如何使此类可串行化？"，如果这一功能被抽象到一个类中，你的回答很简单"从 Serializable 派生出你的类"。你根本无需进一步的解释。

不过，还有一种特殊情况。如果你的类已经是其他一些类的派生类呢？它必须继承于这一个类同时继承于 Serializable。在这种情况下，就可使用多重继承。在 C++中，一个类可以有多个基类。多重继承的语法相当简单：

```
class MultiDerived: public Base1, public Base2
```

例如，假如你对于将 std::string 看作一个 Serializer 知道的简单类型不满意。相反，你宁愿创建一个单独的类型：一个可串行化的字符串。下面是应用多重继承的实现代码：

```
using std::string;
class SerialString: public string, public Serializable
{
public:
  SerialString(string const & str)
    : string(str){}
  void Serialize(Serializer & out)const;
  void DeSerialize(DeSerializer & in);
};
```

Script 52

多重继承在继承抽象类时尤其有用。这种继承处理的是接口而不是实现代码。事实上，正是对多重继承的这一限制也带到了 Java 和 C#中。在这些语言中，你只能继承一个完备的类，但你可以从任何数量的接口给它添加多重继承（等价于 C++的抽象类）。在大部分情况下，这的确是一种非常合理的限制。

第13章

重载运算符 new

在 C++中，new 和 delete 都被认为是运算符。因此，它们几乎可以像其他所有的运算符一样重载。而且与定义一个特定类 operator=一样，你也可以定义特定类运算符 new 和 delete。它们将由编译器自动调用来分配和释放特定类的对象。而且，你可以重载和覆盖 new 和 delete 的全局版本。

13.1 特定类 new

动态内存分配及释放开销并不少。许多程序在释放存储、查找自由块、回收删除的块以及合并碎片上花费了大量时间。如果内存管理是程序的的性能瓶颈，你可以选择多种优化技术加以应用。

基于每个类的重载 new 和 delete 通常用于加速这一特定类的对象的分配和释放。有两种技术可供使用——缓冲和批量分配。

13.1.1 缓冲

缓冲的基本思想是回收比制造开销少。假如我们想加速哈希表的添加，每次添加就分配一个新链接。

在本程序中，链接仅在哈希表被销毁时才释放，而在程序的结尾才会释放哈希表，所以，这不是一个提高性能的十分理想的方法。不过，假如在不同的程序中使用我们的哈希表，而在这些程序中可以选择性地从哈希表中删除条目，或者在程序的生存期可以创建、销毁多个哈希表。在这两种场合，我们均可以加速平均链接分配时间，办法是跟踪当前没有使用的链接。

new1

我们将使用 List::Free 对象作为未用链接的存储地。为了获得一个新链接，我们将调用其

NewLink 方法。为了返回存储池一个链接，我们将调用它的 Recycle 方法。还有一个 Purge 方法用于释放整个存储池。链接的存储池用一个链表来实现。

```
class Link;

class Free
{
public:
  Free(): _p(0){}
  ~Free();
  void Purge();
  void * NewLink();
  void Recycle(void * link);
private:
  Link * _p;
};
```

类 Link 有一个静态成员_freeList，它由重载特定类运算符 new 和 delete 使用（记住，静态成员由这一类的所有实例共享）。

```
class Link
{
  friend class Free;
public:
Link(Link * pNext, int value)
  : _pNext(pNext), _value(value){}

Link *  Next()const { return _pNext; }
int     GetValue()const { return _value; }
// 分配器
void * operator new(std::size_t size)
{
  assert(size == sizeof(Link));
  return _freeList.NewLink();
}
void operator delete(void * mem)
{
  if(mem)
    _freeList.Recycle(mem);
  }
  static void Purge(){ _freeList.Purge(); }
private:
  static   Free _freeList;

  int    _value;
  Link * _pNext;
};
```

请注意，运算符 new 中的断言防止有人对不同的类调用这一特定的运算符。这种情况如

何才能发生呢？就在运算符 new 和 delete 被继承时。如果一个类继承 Link，它没有覆盖这些运算符，对这些派生类调用 new 将返回一个错误大小（基类大小）的对象。

在 List::Add 内创建新 Link 现在被编译器翻译成调用特定类运算符 new，然后调用它的构造函数（如果有的话）。这一方法的优点在于无需对 List 的实现代码作任何修改。

```
List::~List()
{
  while(_pHead != 0)
  {
    Link * pLink = _pHead;
    _pHead = _pHead->Next(); // 取消链接 pLink
    delete pLink;
  }
}

void List::Add(int value)
{
  Link * pLink = new Link(_pHead, value);
  _pHead = pLink;
}
```

一个哈希表包含一个 List 数组，所有 List 在内部针对其链接使用特殊目的的分配器（如图 13-1 所示）。

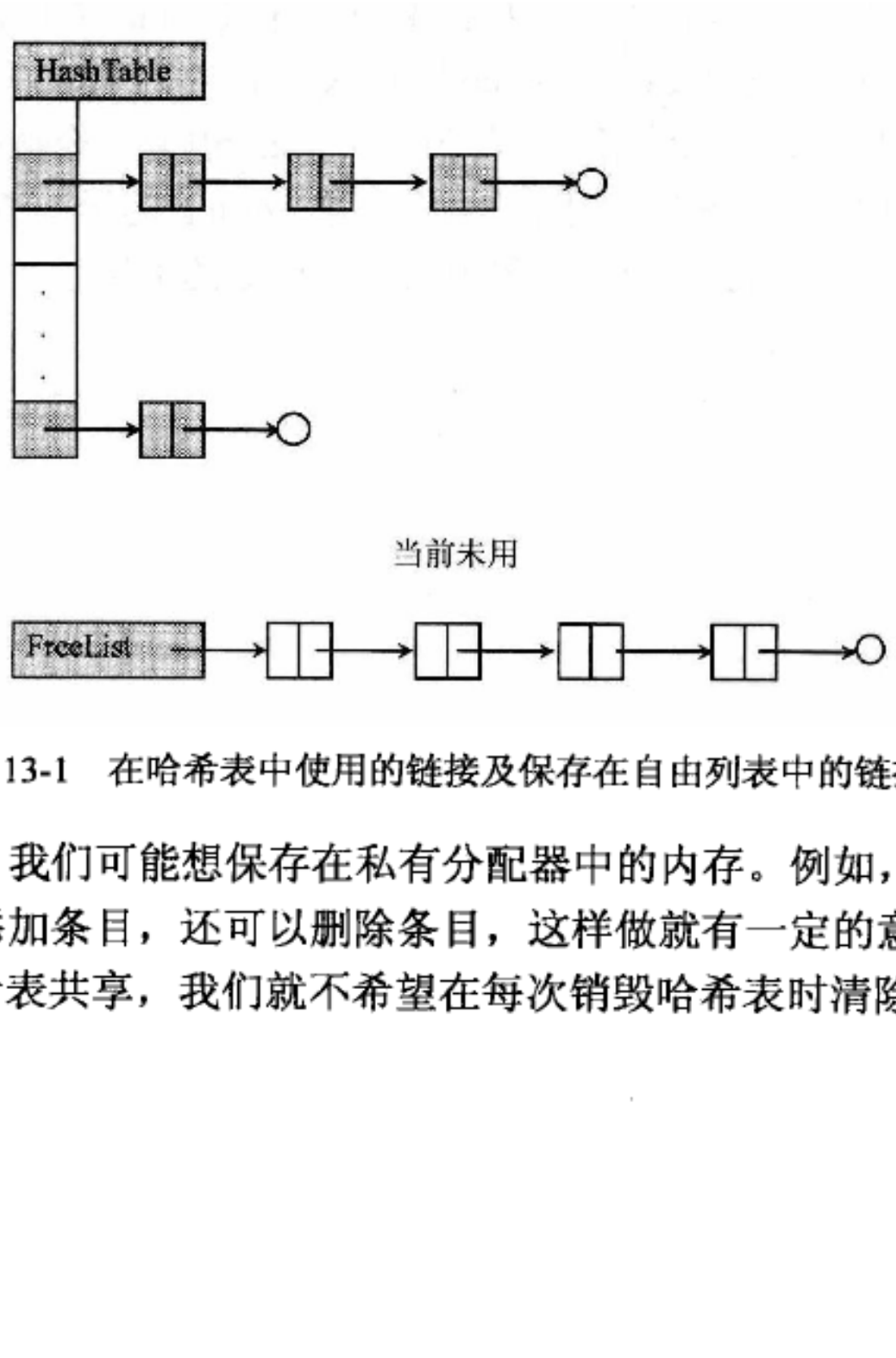


图 13-1　在哈希表中使用的链接及保存在自由列表中的链接

使用完哈希表后，我们可能想保存在私有分配器中的内存。例如，如果程序中只有一个哈希表，它不仅可以添加条目，还可以删除条目，这样做就有一定的意义。相反，如果我们希望链接池由多个哈希表共享，我们就不希望在每次销毁哈希表时清除。

```
class Htable
{
```

```
public:
  explicit HTable(int size): _size(size)
  {
    _aList = new List [size];
  }

  ~HTable()
  {
    delete [] _aList;
    // 释放自由列表中的内存
    List::Link::Purge(); // 可选
  }

  // ...
private:
  List * _aList;
  int    _size;
};
```

注意，Purge 是 Link 的一个静态方法，因而，我们不需要用一个 Link 的实例来调用它。

在实现文件中，我们首先必须定义类 Link 的静态成员_freeList。静态数据自动初始化为 0。

```
List::Free List::Link::_freeList;
```

实现 List::Free 相当简单。我们尽可能从自由列表重用 Link；否则，调用全局运算符 new。因为我们在分配原始内存，我们要请求 sizeof（Link）个字节（char）。当我们删除这一存储空间时，将 Link 转换回它们的原始形式。作为一个 Link 删除一个链接将导致对析构函数的调用（第二次调用）。我们不希望在此删除它，因为这些 Link 的析构函数在调用特定类 delete 时已经被调用了。这里，我们已经位于对象及其析构函数的下层了，现在我们关心的是原始存储区的释放。

```
void * List::Free::NewLink()
{
  if(_p != 0)
  {
    void * mem = _p;
    _p = _p->_pNext;
    return mem;
  }
  else
  {
    //使用全局运算符 new
    return ::new char [sizeof(Link)];
  }
}
```

```
void List::Free::Recycle(void * mem)
{
  Link * link = static_cast<Link *>(mem);
  link->_pNext = _p;
  _p = link;
}

List::Free::~Free()
{
  Purge();
}

void List::Free::Purge()
{
  while(_p != 0)
  {
    // 分配为一个字符数组
    char * mem = reinterpret_cast<char *>(_p);
    _p = _p->Next();
    ::delete [] mem;
  }
}
```

请注意其中我们必须完成的所有强制类型转换。当调用重载 new 时，它应该返回一个空指针。不过在内部，我们既可以从一个链表池回收一个 Link，也可以分配适当大小的原始内存块。我们不希望调用::new Link，因为这样做有一个我们不愿看到的副作用：调用 Link 的析构函数（从运算符 new 返回后，它无论如何都要被调用一次）。

另一方面，通过空指针调用了 delete，因此，为了在列表中保存它，我们必须将它转换成一个 Link。Purge 删除了所有的 Link，就像它们是字符数组一样，因为这是它们的分配方式。同样，我们也不希望将它作为 Link 删除，因为 Link 析构函数已经被调用了。

通常，通过双冒号来区分全局运算符 new 和 delete 调用。它们在这里并不是非用不可，但使用它们可以提高可读性。

13.1.2 批量分配

new2

加速分配的另一个途径是批量分配，然后通过多次调用运算符 new 来分期抵销内存分配的消耗。Link、List 与 HashTable 的实现与前面相同，只是在 List::Free 的位置使用了一个新类 LinkAllocator。它的接口与 List::Free 相同，但它的实现过程更加复杂。除了记住回收 Link 的列表外，它还有一个链接块的单独列表。每个块包含一个类 Block 头以及 16 个连续原始内存片断组成的块，每块的大小为一个 Link 的大小（如图 13-2 所示）。

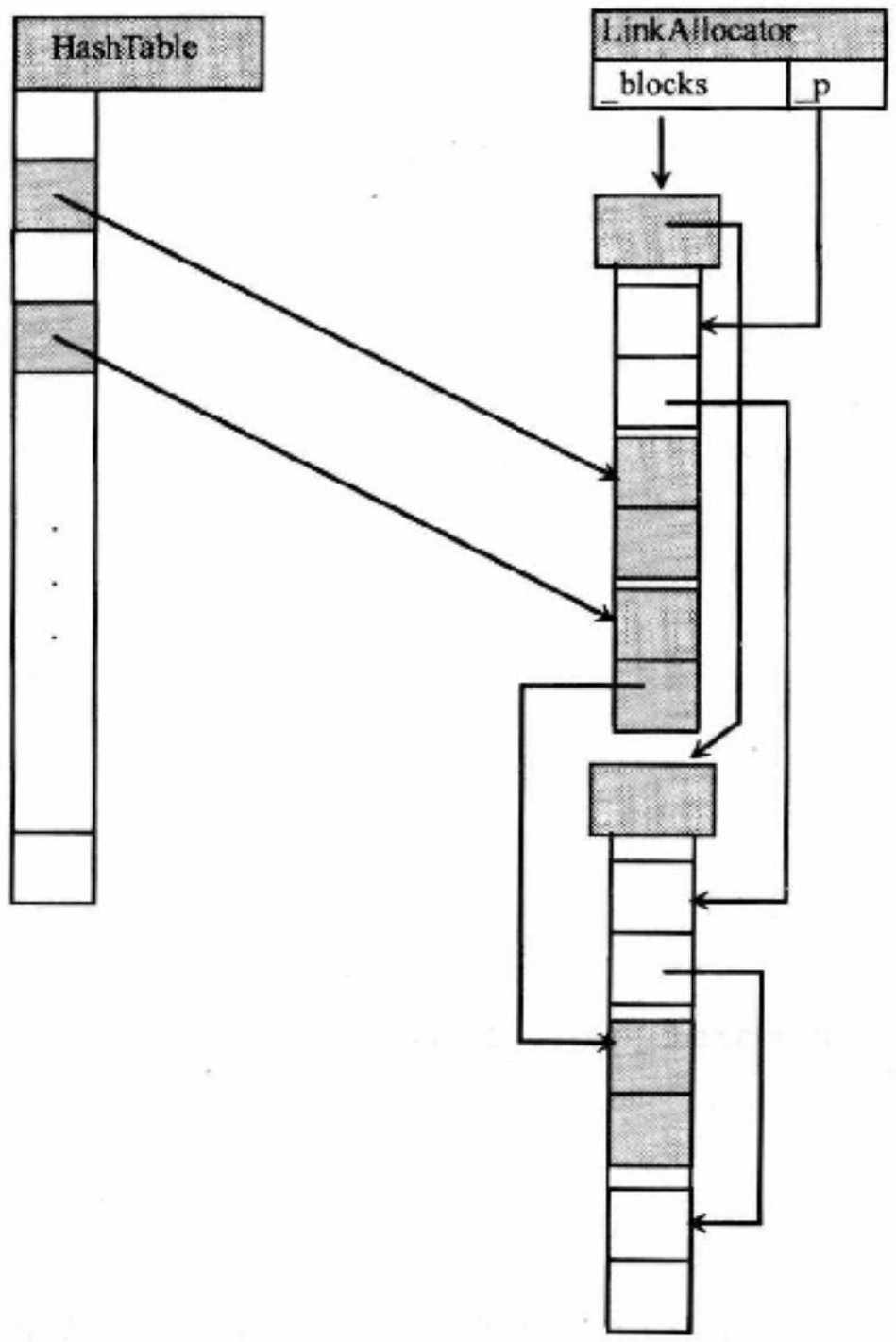


图 13-2　内存块被分割到链接中。一些链接位于自由列表中，另一些被哈希表使用

```
class LinkAllocator
{
  enum { BlockLinks = 16 };
  class Block
  {
  public:
    Block * Next(){ return _next; }
    void SetNext(Block * next){ _next = next; }
  private:
    Block * _next;
  };
public:
  LinkAllocator(): _p(0), _blocks(0){}
  ~LinkAllocator();
  void Purge();
  void * NewLink();
  void Recycle(void * link);
private:
  List::Link  * _p;
  Block * _blocks;
};
```

创建一个新 Link 的方式：

```
void * LinkAllocator::NewLink()
{
  if(_p == 0)
  {
    // 使用全局运算符 new 来分配 link 块
    char * p = ::new char [sizeof(Block)
                  + BlockLinks * sizeof(List::Link)];
    // 添加到块列表中
    Block * block = reinterpret_cast<Block *>(p);
    block->SetNext(_blocks);
    _blocks = block;
    // 添加到链接列表中
    p += sizeof(Block);
    for(int i = 0; i < BlockLinks; ++i)
    {
     List::Link * link =
                reinterpret_cast<List::Link *>(p);
      link->SetNext(_p);
     _p = link;
     p += sizeof(List::Link);
    }
    }
    void * mem = _p;
    p = _p->Next();
    return mem;
  }
```

第一个代码块处理 Link 列表中没有无用链接的情形。16 个（BlockLink）Link 大小的内存块一次性分配，同时为 Block 头分配一定的空间。Block 立即与块的列表链接起来，然后分割成单独的 Link，这些 Link 被添加到 Link 列表。一旦补充了 Link 列表，我们可以从中挑出一个 Link 并放弃。Recycle 的实现与前面相同——即将链接返回给 Link 列表。另一方面，Purge 执行整个块的释放。

```
void LinkAllocator::Purge()
{
  while(_blocks != 0)
  {
    // 分配为一个字符数组
    char * mem = reinterpret_cast<char *>(_blocks);
    _blocks = _blocks->Next();
    ::delete [] mem;
  }
}
```

16 次对 new Link 的调用，只有其中的一次导致实际的内存分配；其他调用只是很快地从列表中挑出一个现成的 Link。

13.1.3　数组 new

即使类 Link 重载了它的运算符 new 和 delete，如果你分配整个 Link 数组（如 new Link [10]），编译器将调用全局 new 来为它分配内存。它不会调用特定类重载。原因有两个，其中一个原因是效率问题。另一个原因是因为 Link 数组和 Link 为不同的类型。一个 Link 的数组不是一个 Link，所以，Link 指定运算符 new 不会应用于一个 Link 的数组。

相反，删除这样一个数组将导致对全局运算符 delete 的调用——而不是它的特定类重载。因为在我们的程序中，我们从未分配 Link 的数组，我们也就不必为此担心。即使我们这样做了，全局 new 和 delete 也会正确执行。

不过，在极少情况下，你确实需要控制数组分配，C++也提供了这样的途径。它可以让你重载运算符 new[]和 delete[]。语法、信号与 new 和 delete 的直接重载类似。

```
void * operator new [](size_t size);
void operator delete [](void * p);
```

唯一的区别是，由编译器传递给 new[]的大小考虑了整个数组加上另外一些编译器用以区分对象指针与对象数组的数据的大小（例如，编译器必须知道数组中的元素数量，以便在调用 delete[]时调用析构函数）。

所有的 4 个运算符（new、delete、new[]以及 delete[]）都被看作重载它们的类的静态成员（也就是说，它们不能访问 this）。

13.2　全局 new

与特定类 new 不一样，全局 new 通常为了调试的目的被重载——当没有定义 NDEBUG 的时候。不过，在某些场合，你可能想永久性重载全局 new 和 delete，因为你有一个更好的分配策略或想要对它进行更多的控制。

在任何情况下，你都可以选择覆盖全局 new 和 delete，或添加自己的特殊版本（遵循的语法稍有不同）。标准运算符 new 带有一个 size_t 类型的参数；标准 delete 带有一个 void *类型的参数。你可以自定义 new 和 delete 版本，让它们带有另外的任意类型参数。例如，你可以定义：

```
void * operator new(size_t size, char * name);
void operator delete(void * p, char * name);
and call the special new using this syntax:
Foo * p = new("special")Foo;
```

不幸的是，无法显式调用特殊的 delete，因此必须要确保标准 delete 能使用特殊 new 正确处理内存分配（否则就永远不会对这样的对象调用 delete）。

那么，如何使用带特殊参数的重载 delete 呢？实际上只在一种场合才会调用它——构建对象期间抛出一个异常时。你可能会想起来，在语言中有一个契约，如果在构建对象期间发生异常，为此对象分配的内存将自动释放。在构建对象期间发生这种情况时，编译器仍然知

道分配内存时所调用的运算符 new 的版本。因此，它能在异常抛出时调用对应版本的 delete。

构建完成后，这一信息消失，编译器无法猜出给定对象适用于哪一个全局 delete 版本。

一旦你定义了一个 new 重载版本，你就可以通过指定它的额外参数显式调用它。或者利用宏替换将代码中的所有 new 调用替换成重载版本。

宏

本书还没有真正谈到宏（macro），它们是标准 C++的一部分，但是并不推荐使用它们。在以前，它们用于代替 C++中的一些复杂功能，如内联函数与模板。现在，已经有更好的途径实现相同的功能，所以，宏很快就被淘汰了。但是为了本书讨论的完整性，我还是解释一下它们的工作原理。

> “由于诸多原因，宏令人讨厌、臭名远扬、不受欢迎，大部分宏作为文本替换的工具，它在应用 C++语法及语义规则之前的预处理中发挥作用。”
>
> ——*Herb Sutter*，发表于 comp.lang.c++.moderated 新闻组

宏通过字面替换来工作。你可以将宏扩展想象成在进入解析 C++语法的主要工作之前，由编译器执行的单独过程。事实上，在老编译器中，宏扩展由一个单独的程序完成，即（宏）预处理程序。

有两种主要的宏类型。第一种类型在逻辑上的代码范围内用一个字符串替换另一个字符串（“逻辑上”的意思是，如果宏定义于一个包含文件，它也就在包含它的文件内发挥作用，依此类推）。

让我举一个实际中可能有用的例子加以说明。如下所示，在文件 DebugNnew.h 中定义下面的宏：

```
#define new new(__FILE__, __LINE__)
```

这一宏将把逻辑上与以上定义对应代码中的所有 new 替换成字符串 new（__FILE__, __LINE__）。而且，宏预处理程序将用发现它的源文件的全名替换所有的特殊预定义符号__FILE__，同时用适当的行号替换所有的__LINE__。于是，如果有一个文件 c:\test\main.cpp，它的内容为：

```
#include "DebugNew.h"
int main()
{
  int * p = new int;
}
```

预处理后的代码如下：

```
int main()
{
  int * p = new("c:\test\main.cpp", 4)int;
}
```

现在，你可以使用自定义重载运算符 new，例如，用它来跟踪所有的内存分配。下面是这样一个简单的例子。

```
void * operator new(size_t size,
                    char const * file,
                    int line)
{
  std::cout << file << ":" << line << std::endl;
  return ::new char [size];
}
```

注意，我们必须确保宏没有包含在定义这一特殊重载的文件中。否则，出现“new”的两个地方都将被“new（__FILE__, __LINE）,”替换，并得到错误代码。

第二种类型宏也用于文本替换，但它更像一个内联函数——它带有参数。同样，因为宏扩展在 C++编译器之外发挥作用，所以，没有类型检查，也就可能出现无法预料的副作用。一个典型的例子是 max 宏：

```
#define max(a, b)(((a)>(b))?(a):(b))
```

注意其中大量的小括号，这是宏的一个特征。程序员要学会在宏参数周围放置小括号，因为它们可能由包含低优先级运算符的表达式所替换。例如，执行下面调用后将发生什么？

```
c = max(a & mask, b & mask)
```

在 max 定义中参数的边上若没有小括号，预处理程序将把它扩展成：

```
c = a & mask >  b & mask? a & mask: b & mask;
```

根据运算符优先级规则，它将被解释成：

```
c =(a &(mask > b)& mask)?(a & mask):(b & mask);
```

（运算符>与操作数的结合比位与“and”要强）这一计算的结果很可能是错的。

当你调用的宏有一个带有副作用的表达式时，情况会更糟。例如，将 max（a11, b11）扩展成：

```
(((a++)>(b++))?(a++):(b++))
```

其中一个变量将增加两次，另一个增加一次。这大概不是程序员所期望的。

还有一件事顺便说一下——请注意在宏定义的 max 与放置参数列表的小括号间没有空格：

```
#define max(a, b)...
```

这种情况在 C++中很少见，标识符与标识符号间的空格改变了结构的意义。没有空格，这是一个包含参数 a 和 b 的宏。有空格，它就是一个无参数宏，而且（a, b）将成为宏体的一部分。

13.3　跟踪内存泄漏

Script 53

通过此技术的一个更有趣应用是跟踪未释放分配（即内存泄漏）。实现的思想是在一个

全局数据结构中保存每个分配的信息并在程序结束时倾倒其内容。重载运算符 new 可以给它添加项，并从此数据结构中删除项。

因为运算符 delete 只能访问指向前一个分配内存的指针，所以我们必须要以适当方式迅速找到基于这一指针的项。我马上会想起通过指针提供线索的映射。我们称这一全局数据结构为 Tracer。

```
class Tracer
{
private:
  class Entry
  {
  public:
    Entry(char const * file, int line)
      : _file(file), _line(line)
    {}
    Entry()
      : _file(0), _line(0)
    {}
    char const * File()const { return _file; }
    int Line()const { return _line; }
  private:
    char const * _file;
    int _line;
};
class Lock
{
public:
  Lock(Tracer & tracer)
    : _tracer(tracer)
  {
    _tracer.lock();
  }
  ~Lock()
  {
    _tracer.unlock();
  }
  private:
    Tracer & _tracer;
  };
  typedef std::map<void *, Entry>::iterator iterator;
  friend class Lock;
public:
  Tracer();
  ~Tracer();
  void Add(void * p, char const * file, int line);
  void Remove(void * p);
```

```
  void Dump();

  static bool Ready;
private:
  void lock(){ ++_lockCount; }
  void unlock(){ --_lockCount; }
private:
  std::map<void *, Entry> _map;
  int _lockCount;
};
```

我们已经定义了两个辅助类：一个是 Tracer::Entry，它用作映射的值；另一个是 Tracer::Lock，它用于临时取消跟踪。它们在实现 Tracer::Add 和 Tracer::Remove 时使用。

方法 Add 为映射添加一个新项，但是仅在激活跟踪时才执行。请注意，它在访问映射时取消跟踪——我们不希望在映射代码内跟踪分配。

```
void Tracer::Add(void * p, char const * file, int line)
{
  if(_lockCount > 0)
    return; // 不要跟踪
  Tracer::Lock lock(*this);
  _map [p] = Entry(file, line);
}
```

方法 Remove 与 Add 一样进行相同的准备，然后搜索删除指针的映射。如果发现了，则删除整个项。

```
void Tracer::Remove(void * p)
{
  if(_lockCount > 0)
    return;

  Tracer::Lock lock(*this);
  iterator it = _map.find(p);
  if(it != _map.end())
  {
    _map.erase(it);
  }
}
```

最后，在程序的结尾，在方法 Tracer 的构造函数中调用方法 Dump 来显示所有泄漏。

```
void Tracer::Dump()
{
  if(_map.size()!= 0)
  {
    std::cerr << "*** Memory leak(s):\n";
    for(iterator it = _map.begin();
        it != _map.end(); ++it)
```

```
      {
char const * file = it->second.File();
      int line = it->second.Line();
      unsigned int addr = reinterpret_cast<unsigned int>(it->first);
      std::cerr << "0x" << std::hex << addr << ": "
        << file << ", line " << std::dec << line
        << std::endl;
    }
    std::cerr << std::endl;
  }
}
```

如果你的C++标准库在终止main后不能直接处理标准输出，请阅读下一节“调试输出”。

因为我们重载了全局运算符new和delete，所以Tracer也必须是一个全局对象。

```
extern Tracer NewTrace;
```

注意，如果在构造函数中有其他一些分配内存的对象，这样做可能导致一些问题。不同文件中全局对象的构建顺序未定义。如果在构建NewTracer之前构建一个内存分配全局对象，我们可能会遇到麻烦。这是我引入了一个静态布尔标志Tracer::Ready的原因，它的原始设置值为false。

```
bool Tracer::Ready = false;
```

Tracer的构造函数将此标志设置成true，而析构函数将它设置回false。

```
Tracer::Tracer()
  : _lockCount(0)
{
  Ready = true;
}

Tracer::~Tracer()
{
  Ready = false;
  Dump();
}
```

实现重载new很简单。

```
void * operator new(std::size_t size,
                    char const * file,
                    int line)
{
  void * p = std::malloc(size);
  if(Tracer::Ready)
    NewTrace.Add(p, file, line);
  return p;
}
```

注意，我们使用的是低级内存分配函数malloc，而不是调用运算符::new。这是因为，我们现时要重载常规较少参数的new。

在终止构建时必须使用对应的重载 delete。

```
void operator delete(void * p,
                     char const * ,
                     int)
{
  if(Tracer::Ready)
    NewTrace.Remove(p);
  std::free(p);
}
```

因为我们使用 malloc 来分配内存，我们必须使用 free 来释放。注意，我没有指定 delete 的最后两个参数。这样可以让你消除编译器警告“Argument not used in function”（参数未被函数使用）。

为了完整性，我们还覆盖了常规全局 new，以防部分代码超出了宏的替换范围（例如，部分 C++标准库）。

```
void * operator new(std::size_t size)
{
  void * p = std::malloc(size);
  if(Tracer::Ready)
    NewTrace.Add(p, "?", 0);
  return p;
}
```

最后，为了跟踪所有的释放，我们必须覆盖全局 delete。

```
void operator delete(void * p)
{
  if(Tracer::Ready)
    NewTrace.Remove(p);
  std::free(p);
}
```

因为我们只希望程序的调试版本能跟踪，所以，我们将宏的定义放在条件编译指令中。头 DebugNew.h 的内容如下。在所有需要测试内存泄漏的源文件中包含这一文件。现在，暂时不要在调用 new 分配数组的文件中包含此文件，除非你同时重载了数组 new 和 delete。（同样，这一宏将干扰对 new 的调用——参见 13.3.2 节）。

```
#if !defined NDEBUG
#include "Tracer.h"
#define new new(__FILE__, __LINE__)
#endif
```

大部分编译器在构建程序的发布（非调试）版本时定义了标志 NDEBUG（no debug 之意）。文件 Tracer.h 其中就包含了重载运算符 new 和 delete 的定义。

同样，我们必须确保重载 new 和 delete 的实现进行有条件地编译。这是因为在编译所有源文件后，由连接器来决定调用哪一个版本的 new 和 delete。如果连接器在代码中没有发现这些运算符的实现代码，它将使用运行时库提供的版本。如果这样的话，它在整个程序中调

用覆盖版本。

最后，我们在 main.cpp 中添加了全局对象 NewTrace 的定义。这一对象的析构函数在 main 结束后倾倒内存泄漏。

```
#if !defined NDEBUG
Tracer NewTrace;
#endif
```

提醒读者一句：我没有谈到线程安全，这超出了本书的讨论范围。不过，你必须意识到，在一个多线程应用程序中，全局运算符 new 可能同时从几个位置调用，所以，你通常必须要考虑一些同步类型（一个临界部分或互斥体）。标准运算符 new（以及 delete）为线程安全，在现在的大部分 C++标准库中是这样的。

13.3.1 调用输出

一个更好的思想重定向倾倒到调试器（这样做的一个额外优点是避免了潜在的库漏洞，这些泄漏可能会阻止 main 后的标准输出）。如果你的程序准备运行于 Windows 平台下，你可以使用函数 OutputDebugString，它在<windows.h>中声明。在调试器中运行此程序时，它将字符串输出到调试窗口。你可以使用 std::stringstream 格式化这些字符串。

```
if(_map.size()!= 0)
{
  OutputDebugString("*** Memory leak(s):\n");
  for(iterator it = _map.begin();
      it != _map.end(); ++it)
  {
    char const * file = it->second.File();
    int line = it->second.Line();
    int addr = reinterpret_cast(it->first);
    std::ostringstream out;
    out << "0x" << std::hex << addr << ": "
    << file << ", line " << std::dec << line << std::endl;
   OutputDebugString(out.str().c_str());
  }
  OutputDebugString("\n");
}
```

如果 C++标准库不进行处理，可以尝试通过使用一个低级转换子程序 itoa（从整数到 ASCII）来避免整数输出。

```
char buffer1 [10];
char buffer2 [8];
out << "0x" << itoa(addr, buffer1, 16)<< ": "
   << file << ", line " << itoa(line, buffer2, 10)<< std::endl;
```

13.3.2 方位 new

C++标准库部分中有一个特殊的重载 new。它称为方位 new（注意，有时 new 的覆盖所

带的额外参数称做方位 new)。这一重载带有一个额外参数：一个空指针。它可在对象的内存已经分配或通过其他途径保存时使用，而且你真正需要的是按某种顺序执行构造函数。空指针参数可以是一个指向某些静态内存或预分配原始动态内存块的指针。方位 new 不分配内存——它使用通过空指针传递给它的内存（由程序员负责确保内存块足够大），而且调用适当的构造函数。它在标准头<new>中声明。

new3

例如，在前一个内存块分配例子中，我们可以使用方位 new 来创建一个应用了这一内存的 Block 对象，它被分配为一个字节数组。通过这一修改，让 Block 的构造函数将指针初始化为指向下一个变得有意义。

```
LinkAllocator::Block::Block(Block * next)
  : _next(next){ }
That would eliminate the need for the method SetNext.
char * p = ::new char [sizeof(Block)
                    + BlockLinks * sizeof(List::Link)];
Block * block = ::new(p)Block(_blocks);
_blocks = block;
```

C++标准库定义了一个对应的方位 delete，它没有做任何事，它在构造函数抛出异常时是必需的。因为方位 new 不分配任何内存，所以删除由它创建的对象是错误的。当然，传递给方位 new 的原始内存必须正确处理。在本例中，用 Purge 方法来释放原始内存。

顺便提及，还有一个数组方位运算符 new[]以及对应的 delete[]。留给读者来练习：请使用它将 Block 头后面的内存转换到 Link 数组为此，你将给 Link 添加哪一类构造函数？

13.4　结论

Script 54

祝贺!通过学习本书，你现在已由一个“周末”型业余程序员升级为一个工业增度型程序员了。你已经学习了一些重要的技术，通过这些技术，你可以为将来构建强健的应用程序作准备。你也熟悉了软件项目开发的工作过程。这是程序员大部分时间要做的工作。

如果你把计算器的原始版本与现在的版本进行比较，你可以发现几乎所有代码行都进行了修改。而且，我们可以一点点地逐步完成这些修改，每一步都得到程序的一个工作版本。这就是一个优秀软件工程的实质所在。

在计算器中仍然存在一个重要的问题。实际上命令行应用程序已无用武之地。程序员专一开发 DOS 程序的时代一去不复返了，当今时代是开辟 GUI 应用的时代。

第三部分

Windows

任何专业程序员都不容忽视 Windows——或者更一般性地讲，不能对基于 Windows 的编程环境一无所知。在开始撰写本书时，我就计划要用一部分内容来介绍 Windows 编程。我曾经用几个章节的篇幅来介绍某多线程应用程序的编写，这一程序用于绘制一个三维动画多边形网，最终我又全部放弃了。

问题在于，讲解 Windows 编程是一个巨大的工程。它需要一本单独的书籍来讲解，甚至需要一个系列来透彻讲解。所以，我在本书中对 Windows 编程的知识进行了非常具有选择性的介绍。我只能着重讨论少数重要的问题，介绍一些有助于读者理解 Windows 编程的基础知识。

第14章 库

从 C++编程方法学的观点来看，Windows 编程完全是一片处女地。一些书籍使用一个 C++有限子集来描述 Windows API（应用程序编程接口，API 包含一套库函数，程序员可以通过它们对 Windows 系统进行操作）。另外还有一些公司开发了他们自己的商业 C++函数库，这些库同样也是基于 C++的某个子集。但是到目前为止，还没有人提出如何使用现代 C++语言处理 Windows 环境下的应用的完整策略。

几年前，我开始在我公司的网站上粘贴一些关于 Windows 编程的小教程和程序片断。这些小程序都是比较简单的东西，只是关于如何封装 Windows 基本功能的一些小技巧。但令我惊奇的是，我的这些小教程在编程社区里受到了热烈欢迎。从访问次数和收到的电子邮件来看，不论是初学者还是有经验的 Windows 程序员都从我的教程里得到了想要的东西。

于是，我开始构思新的计划。从找出一些已有解决方案的不足入手，接着介绍 Windows 编程的范例。我将通过介绍如何编写一些简单的程序来引导你进入 Windows 编程的大门。然后就直奔主题——如何使用 C++中的面向对象范例将 Windows API 以最佳方式封装起来。我没有编写一个完整 Windows 库的雄心壮志，但我会介绍许多关于这方面的技巧。

宏和向导

我们从介绍编写库的禁忌开始。首先，不要去创造一种晦涩的 C++“方言”。这里的“方言”指产生 C++代码的宏系统。例如，你会认为下面这段代码是 C++程序吗？

```
IMPLEMENT_DYNAMIC(CMyThing, CMyObject)
BEGIN_MESSAGE_MAP(CMyThing, CMyObject)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

如果这是用 C++编写 Windows 程序的唯一方法，那么我不得不说 C++是一种不适合

Windows 编程的语言。这样的话，与其用 C++宏预处理程序创造一种新“方言”，还不如彻底发明新的语言——这种语言具有 50%的编程语言特征，但更适合于 Windows 编程。不过，我决定还是先试试 C++再说。

对编写库的程序员来说，另外一种省事的替换机制就是使用所谓的“向导”（wizards）——一个界面友好的、能生成代码的小玩意儿。从原理上说，用 GUI（graphical user interface，图形用户界面）控件生成代码的主意并不坏。如果考虑周详的话，它相当于引入一些更高级编程语言，在此语言的编程中包含对 UI 的控制。如今有很多人在研究“可视化（visual）”语言，这大概是图形化 Windows 应用程序越来越广泛的必然结果。

如果你想发明一种新语言——可视化或者其他类型——那你最好是一个优秀的设计师。编程范例绝不是仅仅把一些杂乱的向导堆砌在一起，这样做通常都是弊大于利。任何编程语言对编程环境最起码的一个要求是不会“丢弃”源代码。

想象一下 C++编译器在生成机器码之后，就把你的源文件删除掉的情形。

而这正是使用向导产生的后果!在向导生成目标 C++代码后，你的输入就丢失了。因为在这种情况下，输入信息由编辑控件中的输入、选中的复选框、点击过的按钮等等组成。这些动作都是由向导实现的“可视化”语言的组成部分。如果这些动作没有被保存下来，你（不管是谁）就不可能对生成的代码作进一步的修改。这时，你不得不重复所有的过程，或者干脆彻底放弃可视化语言，对代码进行手工修改。在这种情况下，向导所有的优势都不复存在。不管这个向导是多么好，它生成的代码都不是友好的。

理想的向导应该是双向的，也就是说，它可以把 C++代码逆向工程转换到可视化形式，不仅能创建，还可以维护可视化代码。但是现在还没有这样的双向向导，我们别无选择，只能应用手边的编程语言——在这里，我们选择了 C++语言。

第15章 编程范例

Windows 程序与其他交互性程序一样，大部分受输入驱动。不过操作系统可以简化 Windows 程序对输入的处理，使程序可以很方便地处理用户的输入。当用户敲击键盘上的某个键或者移动鼠标时，Windows 操作系统可以截取这些事件，对它们进行预处理，然后把它们分配给相应的用户程序。用户程序从 Windows 那里获得所有的消息，它可以对这些消息进行处理也可以不处理它们，对后一种情况来说，就是让 Windows 做应该做的事（缺省处理）。

当一个 Windows 程序启动时，它会在系统中注册一个窗口类（不是 C++类）。通过这个“类数据结构”，程序赋予系统一个叫做窗口过程（window procedure）的回调函数指针。只要 Windows 想要给程序传递消息，通知有趣的事件，它就可以调用这个过程。所谓“回调”指的是：不用去调用 Windows，Windows 会把消息主动告诉你。

程序会在消息循环（message loop）中的每条消息被分配到相应的过程之前把它们扫描一遍。在大多数情况下，消息循环会把消息都返还给 Windows，让 Windows 来做消息分配工作（如图 15-1 所示）。

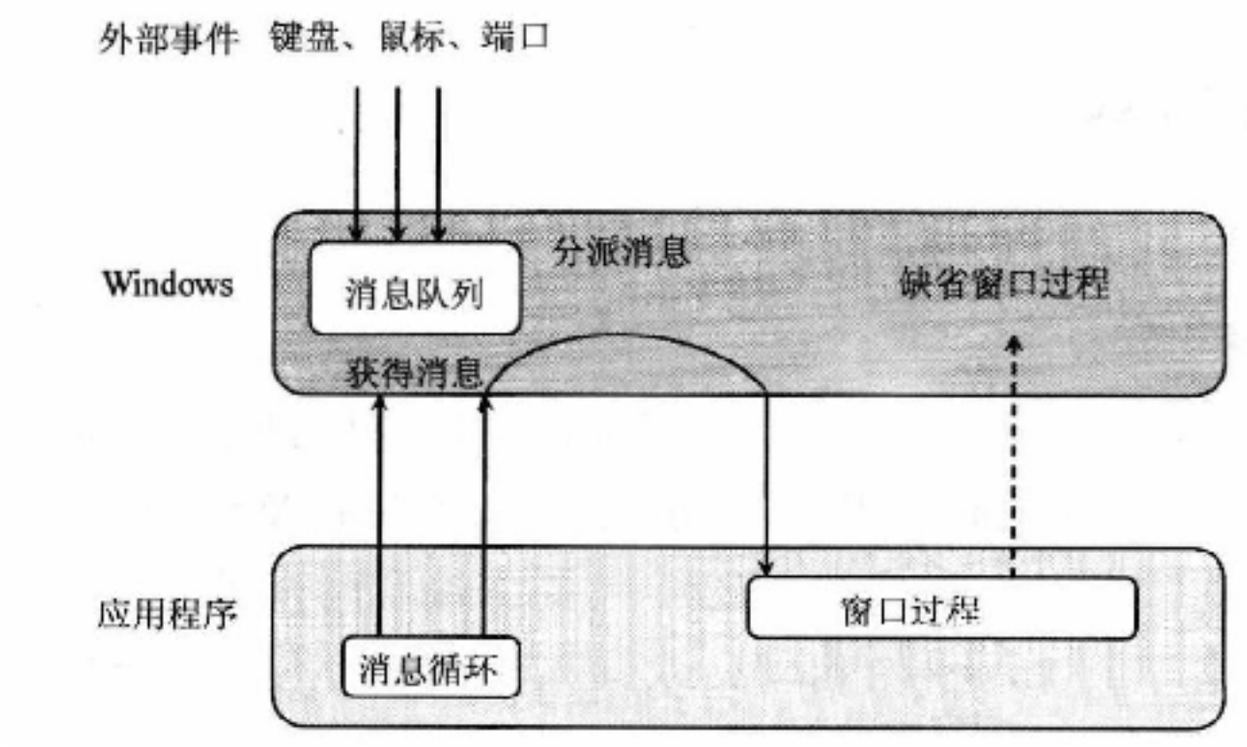


图 15-1 输入驱动 Windows 范例

Windows 是一个多任务操作系统（multitasking operating system）——在同一时刻可以同时运行多个程序。那么，Windows 怎么知道哪个程序需要某条特定消息呢？举例来说，在某一时刻 Windows 截取到一条鼠标消息，那么它通常都会把这条消息分配给创建了鼠标指针所在窗口的程序（除非其他应用程序“捕获”了鼠标）。

大多数 Windows 程序都会在屏幕上创建一个或多个窗口。在某一给定的时刻，在所有窗口中只有一个窗口拥有焦点（focus），这个窗口叫做活动窗口（它的标题栏加亮显示）。键盘消息都发送给拥有焦点的窗口。

虽然也可以由拥有这些窗口的程序来处理事件的消息，但诸如窗口大小改变、最大化、最小化、覆盖和取消覆盖之类事件一般都是由 Windows 来处理的。Windows 程序在运行过程中会收到各种各样的消息，每个程序处理与它有关的那些消息，剩下的消息由 Windows 来处理。

Windows 程序使用 Windows 提供的功能在屏幕上输出文本或图形。Windows 不仅提供了一个高级图形接口，它还把应用程序和实际的图形硬件分离开来。从这一点来说，Windows 的图形功能在很大程度上是与设备无关的。

对 Windows 程序来说，使用标准的 Windows 控件是非常容易的。菜单和消息框都很容易创建。对话框就更常见了——程序员可以使用对话框编辑器设计对话框，使用图标编辑器创建图标。列表框、文本框、滚动条、按钮、单选按钮、复选框等都是 Windows 内置的控件，拿来就可以用，这些特点使得 Windows 编程是具有如此高的吸引力和可用性。

所有的这些功能都可通过 Windows API 函数来实现。Windows API 函数是一个 C 语言函数、类型定义（typedef）、结构和宏的巨大集合，它的声明（直接或间接）包含在<windows.h>中，它的代码通过一套库和 DLL（dynamic-link library，动态链接库）链接到应用程序。

在本章中，我们将使用了一个叫做 Windows 的 Code Co-op。边注列出了对应的脚本代码展开号。

15.1　Hello Windows

Start project windows

我们的第一个 Windows 程序将只创建一个窗口，标题栏显示“Hello Windows!”。虽然很简单，但它还是要比 Kernighan 和 Ritchie 那著名的“Hello World!”程序和我们的第一个“Hello!” C++程序要复杂得多。我们得到的结果要比那看起来象老式电传打字机的输出要丰富得多。我们要创建一个可以到处拖动、改变大小、最小化、最大化与其他窗口层叠的窗口。在窗口的左上角还有一个标准的系统菜单。因此，这已经是一个基本完备的 Windows 程序了！

在 Windows 中，主过程叫做 WinMain。它必须遵循 WINAPI 调用约定，系统根据以下参数来调用它：

- HINSTANCE hInst——程序当前实例的句柄。
- HINSTANCE hPrevInst——此参数是为了保持与 Win 16 的兼容，在 Win 32 中已经不使用。
- LPSTR cmdParam——包含命令行参数的字符串。
- int cmdShow——是否显示主窗口的标志。

你可能注意到了上面的类型名都很奇怪。你必须习惯于这些类型名——Windows 有丰富的类型定义。实际上，在 Windows API 描述中，你几乎看不到 int 或者 char 的类型名。现在，只要知道 LPSTR 实际上是 char *的类型定义就足够了——LPSTR 是 Long Pointer to STRing（指向字符串的长指针）的缩写，此处字符串是一个以 null 终止的数组，“长指针”是 16 位 Windows 时代遗留下来的叫法。

在下面，我会在所有 Windows API 函数的前面加上双冒号（::）。双冒号表明这是一个全局定义函数（不是某个类或名字空间的成员）。这也许有些多余，但这可以使代码的可读性更好。我们稍后将定义 WinClassMaker、WinMaker 及 Window 类。

```
int WINAPI WinMain
 (HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR cmdParam, int cmdShow)
{
 char className [] = "Winnie";
 WinClassMaker winClass(WinProcedure, className, hInst);
 winClass.Register();
 WinMaker maker(className, hInst);
 Window win = maker.Create("Hello Windows!");
 win.Display(cmdShow);
 // 消息循环
 MSG  msg;
 int status;
 while((status = ::GetMessage(& msg, 0, 0, 0))!= 0)
 {
    if(status == -1)
      return -1;
    ::DispatchMessage(& msg);
  }
  return msg.wParam;
}
```

首先，程序创建一个窗口类并对它进行注册。然后创建一个标题为“Hello Windows!”的窗口，并显示该窗口（或者不显示，这取决于给定标志值）。最后，程序进入消息循环，在消息循环中等待从 Windows 传过来的消息（::GetMessage API 函数），并让系统对消息进行分配。当 Windows 调用窗口过程 WinProcedure 时，消息会返回到我们的程序。目前我们暂时不用关心消息循环内部的细节。当::GetMessage 返回 0 时，程序通常终止。最后一个消息的

wParam 包含程序的返回代码。

::GetMessage 函数是一个有趣的三态逻辑例子。它定义的返回数据类型为 BOOL，BOOL 是 int 的类型定义，而不是 bool（布尔）的类型定义（实际上，在 Windows 刚发布的时候，C++中没有 bool 类型，更不要说 C 了）。在帮助文档中说明了它的三种返回类型：非 0、0 和 -1。下面是从帮助文档中摘录的相关内容：

如果函数收到 WM_QUIT 以外的消息，返回值为**非 0**。

如果函数收到的消息为 WM_QUIT，返回值为 **0**。

如果出现错误，返回值为 **-1**。

下面我们介绍 **WinClassMaker** 类，以及如何用它封装 **WNDCLASSEX** 数据结构和**::RegisterClass API** 函数。

```
class WinClassMaker
{
public:
  WinClassMaker(WNDPROC WinProcedure,
                char const * className,
                HINSTANCE hInst);
  void Register()
  {
    if(::RegisterClassEx(&_class)== 0)
      throw "RegisterClass failed";
  }
private:
  WNDCLASSEX _class;
};
```

在 WinClassMaker 的构造函数中，我们把所有的参数都初始化为合理的缺省值。例如，把鼠标的形状设为缺省的箭头形状，将画刷（Windows 用它来绘制窗口的背景）设为缺省的窗口颜色。我们把大多数 Windows 错误翻译成异常（exception）。目前我暂时使用字面字符串作为异常对象，异常对象可以用下面的 catch 子句来捕获：

```
catch(char const * msg)
```

稍后我们将介绍一种更复杂的机制（参见 18.7）。

在注册一个类时，我们必须提供窗口过程的指针、类的名字和拥有类的实例句柄。

```
WinClassMaker::WinClassMaker
 (WNDPROC WinProcedure,
   char const * className,
   HINSTANCE hInst)
{
  _class.lpfnWndProc = WinProcedure;      // 强制性的
  _class.hInstance = hInst;               //强制性的
  _class.lpszClassName = className;       //强制性的
  _class.cbSize = sizeof(WNDCLASSEX);
```

```
_class.hCursor = ::LoadCursor(0, IDC_ARROW);
_class.hbrBackground = reinterpret_cast<HBRUSH>
                    (COLOR_WINDOW + 1);
_class.style = 0;
_class.cbClsExtra = 0;
_class.cbWndExtra = 0;
_class.hIcon = 0;
_class.hIconSm = 0;
_class.lpszMenuName = 0;
}
```

注意在上面的程序中，我们不得不使用一些憋脚的技巧。WNDCLASSEX 的数据成员 hbrBackground 被定义为 HBRUSH，你可以用 Windows 定义的标准颜色常量来对它进行初始化，我们在这里使用的是 COLOR_WINDOW。这就需要使用显式的类型转换。但不幸的是，颜色常量 COLOR_SCROLLBAR 被定义为 0，这时 Windows 就会误以为是空画刷。所以在上面的程序中，我们把颜色常量加 1 后再传递给 WNDCLASSEX，这样就可以避免颜色常量为 0 的情况。在 Windows 编程中有很多这样的小技巧。

WinMaker 类初始化和保存描述特定窗口的所有参数。

```
class WinMaker
{
public:
  WinMaker(char const * className, HINSTANCE hInst);
  HWND Create(char const * title);
private:
  HINSTANCE   _hInst;          // 程序实例
  char const *_className;      // 窗口类的名字
  DWORD       _style;          // 窗口风格
  DWORD       _exStyle;        // 扩展的窗口风格
  int         _x;              // 水平位置
  int         _y;              // 垂直位置
  int         _width;          // 窗口宽度
  int         _height;         // 窗口高度
  HWND        _hWndParent;     // 父类或拥有者
  HMENU       _hMenu;          // 菜单或者子窗口 ID
  void       *_data;           // 窗口创建数据
};
```

WinMaker 构造函数中带有它的窗口类名字和程序实例的句柄。类名是 Windows 查找本窗口对应窗口过程所必需的。其余的参数都设置为合理的缺省值。例如，由系统确定窗口的位置和大小。WS_OVERLAPPEDWINDOW 是最常用的顶层窗口风格，其左边有标题栏左和系统菜单，右边有最小化按钮、最大化按钮和关闭按钮。它还有一个“粗”边框，可以用鼠标拖动此边框来改变窗口的大小。

```
WinMaker::WinMaker(char const * className,
                   HINSTANCE hInst)
  : _style(WS_OVERLAPPEDWINDOW),
```

```
    _exStyle(0),
        _className(className),
        _x(CW_USEDEFAULT),          // 水平位置
        _y(0),                      // 竖直位置
        _width(CW_USEDEFAULT),      // 窗口宽度
        _height(0),                 // 窗口高度
        _hWndParent(0),             // 父窗口或窗口拥有者
        _hMenu(0),                  //菜单或子窗口标识符
        _data(0),                   //窗口创建数据
        _hInst(hInst)
    {}
```

所有的这些参数都传递给::CreateWindowEx API 函数，::CreateWindowEx API 根据这些参数创建窗口（但尚未显示此窗口）。

```
HWND WinMaker::Create(char const * title)
{
  HWND hwnd = ::CreateWindowEx(
    _exStyle,
    _className,
    title,
    _style,
    _x,
    _y,
    _width,
    _height,
    _hWndParent,
    _hMenu,
    _hInst,
    _data);

  if(hwnd == 0)
    throw "Window Creation Failed";
  return hwnd;
}
```

Create 带有在标题栏上显示的窗口标题，并返回成功创建窗口的句柄。我们可以很方便地把这个句柄封装到一个名为 Window 的类中。这个类没有保存特定窗口的句柄，而是为以后在该窗口中进行操作的众多 Windows API 函数提供一个接口。

```
class Window
{
public:
  Window(HWND h = 0): _h(h){ }
  void Display(int cmdShow)
  {
    assert(_h != 0);
    ::ShowWindow(_h, cmdShow);
    ::UpdateWindow(_h);
```

```
  }
private:
  HWND _h;
};
```

要使窗口可见，必须通过适当的参数调用::ShowWindow，这些参数指定窗口的初始状态是否应该为最小化、最大化，或为缺省大小。调用::UpdateWindow 可以刷新窗口内容。

窗口过程中必须包含以下信号，这是 Windows 的硬性代码：

```
LRESULT CALLBACK WinProcedure
 (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

注意它的调用方式、参数类型和返回值。它们都是在 windows.h 中定义的类型定义。CALLBACK 是与语言无关的预定义调用约定（关于参数推入堆栈的顺序等等）。LRESULT 是一种返回值类型。HWND 是某窗口的句柄，UNIT 是一个用来标识消息的无符号整数，WPARAM 和 LPARAM 是随每条消息传递的两个参数的类型。

当 Windows 每次要向用户程序传递消息时，它就调用 WinProcedure（如图 15-2 所示）。窗口句柄标识哪个窗口应当对消息作出响应。记住：一个窗口过程可以为同一个窗口类的若干个实例服务。每个实例都有它自己的窗口及不同的句柄，不过经历的过程是一样的。

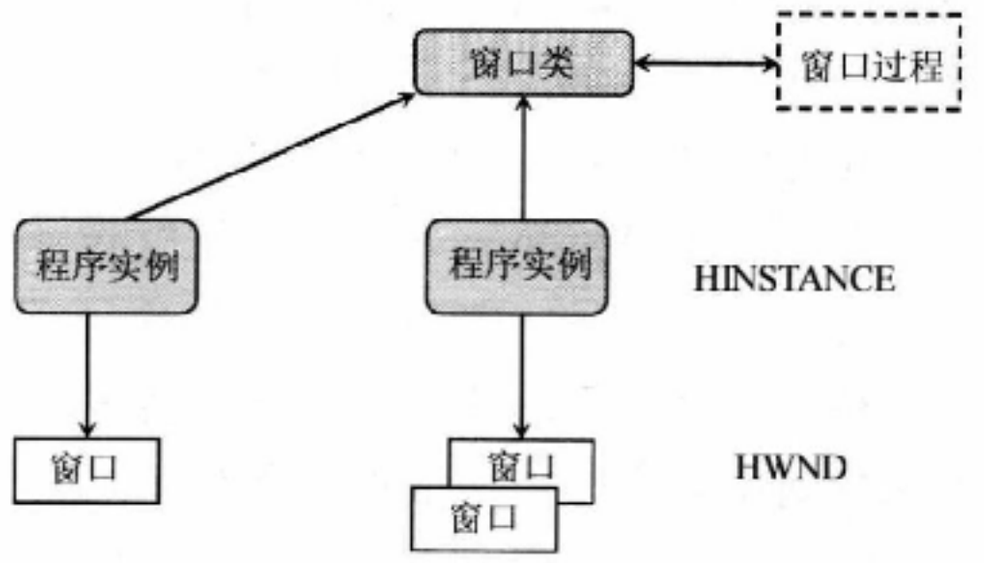


图 15-2 同一程序实例、它们的窗口以及程序窗口类和过程间的关系

消息其实只是一个编号，在 windows.h 中定义了这些编号对应的符号名字。例如通知窗口重绘的消息定义如下：

```
#define WM_PAINT        0x000F
```

每条消息都带有两个参数，参数的含义根据消息的不同而不同（对 WM_PAINT 来说，参数是没有意义的）。

如果你想进一步了解窗口过程和不同类型的消息，请参看编译器提供的帮助文档。

下面是窗口过程的最简单实现：

```
LRESULT CALLBACK WinProcedure
 (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
  switch(message)
  {
  case WM_DESTROY:
    ::PostQuitMessage(0);
    return 0;
  }
  return ::DefWindowProc(hwnd, message, wParam, lParam);
}
```

这个程序只处理了一条消息：WM_DESTROY。这条消息在窗口关闭时传递给程序。这

时窗口已经被关闭了，我们要做的就是终止 WinMain。我们通过发出最终退出消息来终止 WinMain。我们还通过它传递返回码——本例中为 0。这条消息将终止消息循环，并把控制权返还给 WinMain。

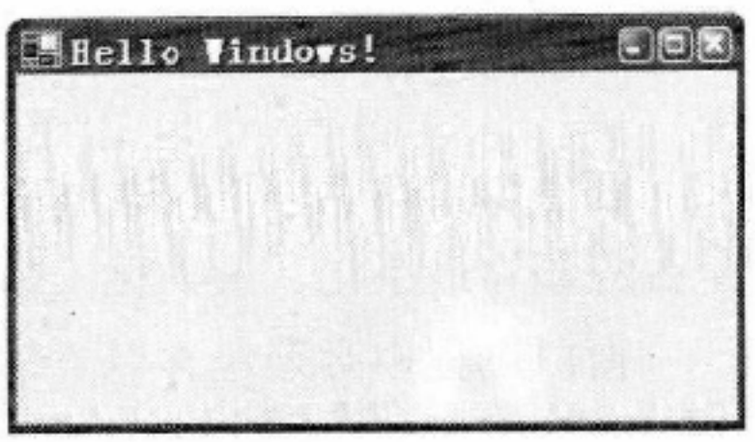


图 15-3　输出 Hello Windows!

图 15-3 示意了以上小应用程序所创建的窗口——可以改变窗口大小的边界、最小化按钮、最大化按钮、关闭按钮、Windows 缺省图标以及点击图标出现的系统菜单。

Script 1

为了使项目更加结构化，我们将把源代码分布在多个文件中。

15.2　封装

许多 Windows API 函数都有很多参数。例如，当你创建一个窗口时，你必须要指定它的位置、大小、风格、标题等等。这就是为什么 CreateWindowEx 有 12 个参数的原因。RegisterClassEx 也有 12 个参数，不过这 12 个参数包含在单独的 WNDCLASSEX 结构里。你可以看到，用 Windows API 函数进行设计时并没有固定的模式。

我们封装这种 API 函数类型的途径是创建一个 C++类，把它的所有参数组合在一起。大部分参数具有合理的缺省值，在构造函数中应该对它们进行正确的初始化。那些没有缺省值的参数则作为实参传递给构造函数。主要思想就是，当创建了一个基于该类的对象后，它不需要修改就可以使用。不过，如果需要修改，也可以调用适当的方法来完成。例如，为了更加完善 WinMaker 类，我们应该加入诸如 SetPosition 和 SetSize 之类的方法，覆盖_x、_y、_width 和_height 的缺省设置。

下面我们分析一下这里的 WinClassMaker 和 WinMaker 类。WinClassMaker 封装了 RegisterClassEx API。RegisterClassEx API 的参数是一个可以直接把它嵌入到类中的结构。10 个参数中的其中 3 个（窗口过程、类名和程序实例）不能使用缺省值，于是它们在构造函数中传递。窗口的背景颜色一般都缺省采用当前系统的设定——即 COLOR_WINDOW 常量。鼠标指针也一般都为系统的缺省箭头形状——即 IDC_ARROW 常量。结构的大小必须设置为 sizeof（WNDCLASSEX）。其他参数都可安全设为 0。在调用 WinClassMaker::Register 之前我们不用做任何其他事情。当然，在更复杂的程序中，我们也许要修改这些设置，为了完成以上目的，我们必定要添加一些方法。关于这一点我们稍后讨论。

同样，CreateWindowEx API 封装在 WinMaker 类里。没有缺省值的参数包括：类名、程序实例和窗口标题。不过，这一次为了创建多个窗口，我们需要多次调用 WinMaker::Create。很可能这些窗口的标题不一样，于是我们把标题作为参数传递给 Create。

小结：程序的主过程创建一个特定类窗口，然后进入消息处理循环。在消息循环中，程

序等待从 Windows 传递过来的消息。一旦消息到达，程序把它返回给 Windows。随后系统用同样的消息回调程序的窗口过程。这时可以手工对消息进行处理，也可以使用系统缺省的处理方式。在程序中必须实现窗口过程，它的地址在注册窗口类时传递给系统。从窗口过程返回后，控制权又回到消息循环，整个处理过程又重新开始。最终系统发送“退出”消息，这一循环才会结束。

第16章 通过C++控制Windows

在我们进一步开发自己的 Windows 库之前，我们先要把重要的命名原则确定下来。你可能已经注意到，在上一章中，所有与 Windows 有关的类的命名前面，我都给它们加一个 Win 前缀。这种命名常常在过去编程时代使用。不过我们现在有了更好的命名机制，我们不仅要引入而且还要强制执行这一命名约定。这就是我们下面要介绍的名字空间（namespace）。

Script 2

很容易就能把与 Windows 有关的类、模板和函数的定义包含在一个名字空间里，为了方便起见我们把这个名字空间叫做 Win。于是，对使用库的用户来说，输入 Win::Maker 而不用输入 WinMaker，输入 Win::Dow 而不用输入 Window[1]，如此等等。这种方式还可以让我们在用户自定义名字中自由使用 Win 前缀（例如，库的客户仍然可以将他们的自定义类叫做 WinMaker，而不用担心与我们库中的类名发生冲突）。

Script 3

Win::ClassMaker 和 Win::Maker 类都有各自的一对头/实现文件。因为我们计划将 Win::Procedure 函数也将作为库的一部分，所以它也有自己的头/实现文件对。

16.1 模型—视图—控制器

Windows 编程的最大挑战在于，如何使构成窗口过程核心的大段 swith 语句看起来不是那么丑陋。我们将一步一步地解决这个问题。

[1] Win::Dow 的命名似乎有些奇怪。其次就是 Win::H 和 Win::Handle，但更多的是反映其实现逻辑而不是功能性（它是一个句柄）。命名有时很难确定，尤其在封装现有的接口时。

你必须了解 Windows API 函数的基本结构是在若干年前建立的。那时的计算机速度比现在的计算机速度慢了若干数量级，使用 C 语言代替汇编语言进行开发是一种风险，而且并没有多少软件公司愿意承担这种风险。C++甚至还没有发明，强类型还只是一种超前的思想。对于塞在对应的 case:标签语句下面的代码块，swith 语句成为最快的消息分配办法。至于程序是否可靠、是否有毛病并不重要，重要的是程序的速度要快。用户输入与屏幕响应之间的反馈循环必须尽可能的简短，否则计算机将显得行动迟缓、反应迟钝。

今非昔比，处理器的主频已经超过了 1GHz——每秒有 10 亿个时钟周期[1]。这就意味着在用户移动鼠标到屏幕上的光标随之移动的这段时间里，计算机可以轻松执行成百上千条指令。用户输入的处理——决定做出什么样的响应已经不再是瓶颈了。

第一步就是要把程序的用户界面和做实际工作的引擎区分开来。传统上与 UI(用户界面)无关部分的内容叫做模型，与用户界面相关的内容分成两部分：视图和控制器。

视图的功能是把数据显示给用户。程序的这一部分就是在程序窗口中绘制图片和显示文字。视图通过模型得到要显示的数据。

控制器的功能是接收并解释用户的输入。当用户输入文本、点击鼠标或者选择菜单项时，控制器是第一个收到通知的。控制器把原始输入转换成模型可以理解的形式。通知模型后，控制器同时调用视图来刷新显示（在某些更复杂的实现方法中，模型可能选择性地通知视图所作的变化）。

模型—视图—控制器范例的功能非常强大，我们封装 Windows 时肯定要应用它。这里的技术问题是当一个全局窗口过程接收到用户输入后，如何通知适当的控制器。第一种尝试是同时让 Controller 成为全局对象，于是窗口过程可以访问全局控制器。不过请记住，在一个程序中可能有多个窗口和窗口过程。某些窗口过程可以被多个窗口共享。我们需要做的是在控制器和窗口句柄之间建立一个映射。每条窗口消息提供一个窗口句柄 HWND，它唯一确定了消息发送的目的地。



Script 4

窗口到控制器的映射有许多实现方式。例如，可以设置一个全局映射进行翻译工作。不过，另外还有一种更简单的方法——我们可以把一个控制器的指针存储在 Windows 的内部数据结构中。Windows 为每个窗口保留一个独立的数据结构，每个数据结构中都可以存储一个用户自定义的 long。每当我们创建一个新窗口，我们就可以创建一个新的 Controller 对象，并让 Windows 存储它的指针。每当窗口过程接收到一条消息，我们就可以从随消息发送过来的窗口句柄中取回这个指针。

对 Windows 内部数据结构进行设置和访问的 API 函数分别为 SetWindowLong 和 GetWindowLong。你必须指定要访问的是内部数据的所在窗口，以及要指定访问哪个长型数

[1] 译者注：现在的 CPU 主频速度已经超过了 2GHz。

据——在窗口内部数据结构中有若干种预定义长型数据，有些可以在创建窗口时同时添加。通常我们都把控制器指针存储在名为 GWL_USERDATA 的长型数据中。每个窗口甚至按钮和滚动条（按钮和滚动条本质上也是窗口）的内部数据结构中都有这个长型数据。从名字我们就可以看出，GWL_USERDATA 中存储的是用户自定义的数据。使用长型数据是因为指针和长型整数的大小一样——至少在 32 位 Windows 中是一样的。我不知道在 64 位的 Windows 中是不是还是这样，但我觉得应该如此。如果不是，那么 64 位的 Windows API 函数就与 32 位的 Windows API 函数有天壤之别了。

在使用 Get/SetWindowLong API 函数时，还有一个小问题：它们是无类型的。它们的输入和返回值都是长型整数，但这不是我们所希望的。我们更希望得到类型安全（type-safe）的数据。为了实现这一点，我们把这两个函数都封装在模板中，用存储的数据的类型把它参数化。

```
namespace Win
{
  template <class T>
  inline T GetLong(HWND hwnd, int which = GWL_USERDATA)
  {
return reinterpret_cast<T>
        (::GetWindowLong(hwnd, which));
 }

  template <class T>
  inline void SetLong(HWND hwnd, T value,
                      int which = GWL_USERDATA)
  {
::SetWindowLong(hwnd, which,
                    reinterpret_cast<long>(value));
 }
}
```

实际上，如果你用的编译器支持成员模板，你可以自己创建 Win::Dow 类的 GetLong 和 SetLong 方法。成员模板是由某一种类型参数化的方法。一个类可以是它自己的模板，由其他数据类型参数化。（auto_ptr 模板就有这样的成员模板，参见 10.8 了解更详细的信息）。

```
namespace Win
{
  class Dow
  {
  public:
Dow(HWND h = 0): _h(h){}
template <class T>
inline T GetLong(int which = GWL_USERDATA)
{
   return reinterpret_cast<T>(::GetWindowLong(_h, which));
}
```

```
template <class T>
inline void SetLong(T value, int which = GWL_USERDATA)
{
  ::SetWindowLong(_h, which,
                  reinterpret_cast<long>(value));
}
void Display(int cmdShow)
{
  assert(_h != 0);
  ::ShowWindow(_h, cmdShow);
  ::UpdateWindow(_h);
}
  private:
HWND _h;
};
}
```

注意参数 which 缺省值的用法。如果在调用这些函数时，没有最后一个参数，那么它的缺省值为 GWL_USERDATA。

下面我们准备创建一个实现窗口过程的程序片段。总是在程序的开始，从窗口句柄取回指向控制器对象的指针。

```
LRESULT CALLBACK Win::Procedure(
            HWND hwnd,
            UINT message,
            WPARAM wParam,
            LPARAM lParam)
{
  Win::Dow win(hwnd);
  Win::Controller * pCtrl = win.GetLong<Win::Controller *>();
  switch(message)
  {
    case WM_NCCREATE:
    {
      Win::CreateData const * create =
        reinterpret_cast<Win::CreateData const *>(lParam);
      pCtrl = static_cast<Controller *>(
        create->GetCreationData());
    pCtrl->SetWindowHandle(hwnd);
    win.SetLong<Win::Controller *>(pCtrl);
    break;
}
case WM_DESTROY:
  // 已不在屏幕上
  pCtrl->OnDestroy();
  return 0;
case WM_MOUSEMOVE:
```

```
      {
        POINTS p = MAKEPOINTS(lParam);
        Win::KeyState kState(wParam);
        if(pCtrl->OnMouseMove(p.x, p.y, kState))
          return 0;
      }
    }
    return ::DefWindowProc(hwnd, message, wParam, lParam);
  }
```

我们在第一批发送到窗口的某条消息中对 GWL_USERDATA 进行初始化。这条消息就是 WM_NCCREATE（Non-Client Create，非客户创建之意），它是在窗口的非客户部分（边界、标题栏、系统菜单等）创建之前传送过来（在 WM_NCCREATE 之前还有另一条消息：WM_GETMINMAXINFO，不过，对它的处理超出了本书的讨论范围）。

我们存储在 GWL_USERDATA 中的指针作为创建窗口的数据传递给 WM_NCCREATE 消息。程序中我们还使用了 Win::CreateData 类，它是一个 Windows 数据结构的小型封装。因为我们想把 Windows 传递给我们的指针从 CREATESTRUCT 转换成 Win::CreateData，所以，我们必须使用继承而不能用嵌入（既可以继承于一个结构，也可以继承于一个类）。

```
namespace Win
{
  class CreateData: public CREATESTRUCT
  {
  public:
    void * GetCreationData()const
      { return lpCreateParams; }
    int GetHeight()const { return cy; }
    int GetWidth()const { return cx; }
    int GetX()const { return x; }
    int GetY()const { return y; }
    char const * GetWndName()const { return lpszName; }
  };
}
```

下面我们对窗口的创建过程进行详细地分析。CreateWindowEx API 函数（从 Win::Maker::Create 中调用）对 Windows 的内部数据结构进行初始化，接着先用 WM_GETMINMAXINFO 消息调用我们的窗口过程（我们忽略了此消息）；然后又用 WM_NCCREATE 消息调用窗口过程。此时，GWL_USERDATA 的值还是 0，但在处理 WM_NCCREATE 消息的过程中我们对它进行了初始化。接下来所有对窗口过程的调用都可以从 GWL_USERDATA 中得到控制器指针。

对顶层窗口来说，WM_DESTROY 消息非常重要。一般“退出”命令都在这里发出。在 WM_DESTROY 之后还有一些消息会发送给窗口，其中最常见的是 WM_NCDESTROY 消息，不过，在这里我们暂时忽略这些消息。

在程序中我还加入了对 WM_MOUSEMOVE 消息的处理，目的是演示消息处理器的设计

思想。当鼠标移动时，WM_MOUSEMOVE 消息就被发送到窗口。在一般的窗口过程中，我们总是把消息中的参数取出来，传递给相应的处理器——控制器的某一方法。

WM_MOUSEMOVE 消息与三个参数相关联：x 坐标、y 坐标以及控制键和按钮的状态。x 坐标和 y 坐标打包在一个 LPARAM 中，Windows 有一个宏 MAKEPOINTS 可以很方便解开这个包。宏 MAKEPOINTS 可以把 lPARAM 转换成一个 POINTS 结构。然后用户从 PONITS 结构中读取 x 坐标和 y 坐标的值，并传递给处理器。

控制键和按钮的状态当作一组位在 WPARAM 内进行传递。在程序中可以通过 MK_CONTROL、MK_SHIFT 之类的掩码对这些位进行访问，这些掩码都是由 Windows 定义的。我们会把这些位操作封装在 Win::KeyState 类中。

```
namespace Win
{
  class KeyState
  {
  public:
    KeyState(WPARAM wParam): _data(wParam){}
    bool IsCtrl()const
      { return(_data & MK_CONTROL)!= 0;}
    bool IsShift()const
      { return(_data & MK_SHIFT)!= 0; }
    bool IsLButton()const
      { return(_data & MK_LBUTTON)!= 0; }
    bool IsMButton()const
      { return(_data & MK_MBUTTON)!= 0; }
    bool IsRButton()const
      { return(_data & MK_RBUTTON)!= 0; }
  private:
    WPARAM    _data;
  };
}
```

Win::KeyState 方法的返回值为 control 键、shift 键以及鼠标左、中、右键的状态。例如，如果你在移动鼠标的过程中同时按下鼠标的左键和 shift 键，则 IsLButton 和 IsShift 的值都为真。

在创建窗口的 WinMain 中，我们对控制器进行初始化，并把它和窗口的标题一起传递给 Win::Maker::Create。TopController 是一个顶层窗口 Win::Controller 的简化实现例子。

```
TopController ctrl;
win.Create(ctrl, "Simpleton");
```

这是修改后的 Create。它把 Controller 的指针作为窗口创建数据用户自定义部分（最后一个参数）传递给 CreateWindowEx。

```
HWND Win::Maker::Create(Win::Controller & controller,
                        char const * title)
{
```

```
    HWND hwnd = ::CreateWindowEx(
      _exStyle,
      _className,
      title,
      _style,
      _x,
      _y,
      _width,
      _height,
      _hWndParent,
      _hMenu,
      _hInst,
      &controller);

    if(hwnd == 0)
      throw "Internal error: Window Creation Failed.";
    return hwnd;
  }
```

小结：客户创建控制器并把它传递给 Win::Maker 类的 Create 方法。在 Create 方法中，控制器的指针被加入到窗口创建数据中，并被 Windows 作为 WM_NCREATE 消息的参数进行传递。窗口过程解包窗口创建数据，并把控制器指针存储在窗口内部数据结构的 GWL_USERDATA 中。在随后的消息处理过程中，窗口过程从这个数据结构中读取控制器，调用相应的方法来处理消息。最后，作为 WM_DESTROY 消息的响应，窗口过程最后一次调用控制器。

16.1.1　控制器

现在我们已经搞清楚了控制器的传递过程，下面我们介绍控制器的实现。我们的目标是把一个窗口的所有逻辑过程都集中到一个类中。做一个通用的窗口过程来负责这一切——庞大的 switch 语句、消息参数的解包和再打包，以及把消息传递到缺省的窗口过程。一旦消息通过 swith 语句块传递，就需要用正确的（强类型）参数去调用相应的 Controller 方法。

现在我们只创建一个控制器程序片段。最终我们会给它加上很多方法——有多少消息就有多少方法。

控制器保留它所服务窗口的句柄，此句柄在处理 WM_NCCREATE 消息时在窗口过程中初始化。这也是我们为什么把 Win::Procedure 作为 Win::Controller 友元的原因。因为由它衍生的类需要对它进行访问，所以句柄本身是受保护类型(protected)，而不是私有类型(private)。此时只有两个消息处理器方法：OnDestroy 和 OnMouseMove。

```
namespace Win
{
  class Controller
  {
    friend LRESULT CALLBACK Procedure(HWND hwnd,
```

```
                    UINT message, WPARAM wParam, LPARAM lParam);
  void SetWindowHandle(HWND hwnd){ _h = hwnd; }
public:
  virtual ~Controller(){}
  virtual bool OnDestroy()
    { return false; }
  virtual bool OnMouseMove(int x, int y, KeyState kStat)
    { return false; }
protected:
    HWND  _h;
  };
}
```

记住 Win::Controller 将会成为我们 Windows 库的一部分——它将作为所有用户自定义控制器的基类，这也就将所有消息处理器声明为 virtual 的原因。布尔型返回值的含义是“我已经处理了那条消息，不必再调用 DefWindowProc 了”。不过在我们的程序中没有实现对消息的处理，所以返回值总是 false。

Windows 库的客户应当自定义一些继承于 Win::Controller 的控制器，还要覆盖某些消息处理器。在这里，唯一需要覆盖的消息处理器是 OnDestroy——它必须通过发送“退出”消息来关闭应用程序。它的返回值是 true，这样就不会调用缺省的窗口过程。

```
class TopController: public Win::Controller
{
public:
  bool OnDestroy()
  {
    ::PostQuitMessage(0);
    return true;
  }
};
```

小结：我们设计库的目的是让用户需要做的工作最少化，并避免犯一些常规的错误。用户对每一个 Windows 类都需要定制一个控制器，这些控制器可以从我们的库类 Win::Controller 继承。用户只对那些需要非缺省实现过程的方法进行实现（或覆盖）。因为用户具有所有这些方法的原型，所以没有误解消息参数的危险。解释和解包部分在一般性的 Win::Procedure 中完成。Win::Procedure 的代码我们已经一次性完成，而且通过了全面测试。

16.1.2 客户代码

这部分程序由使用我们库的客户编写。虽然这部分代码很简单，但稍后我们还会进一步简化它（见 Script 14）。

```
#include "Class.h"
#include "Maker.h"
#include "Procedure.h"
```

```
#include "Control.h"
#include "Window.h"

class TopController: public Win::Controller
{
public:
  bool OnDestroy()
  {
    ::PostQuitMessage(0);
    return true;
  }
};
int WINAPI WinMain
 (HINSTANCE hInst, HINSTANCE hPrevInst,
    LPSTR cmdParam, int cmdShow)
{
  char className [] = "Simpleton";
  Win::ClassMaker winClass(className, hInst);
  winClass.Register();
  Win::Maker maker(className, hInst);
  TopController ctrl;
  Win::Dow win = maker.Create(ctrl, "Simpleton");
  win.Display(cmdShow);

  MSG  msg;
  int status;
  while((status = ::GetMessage(& msg, 0, 0, 0))!= 0)
  {
    if(status == -1)
      return -1;
    ::DispatchMessage(& msg);
  }
  return msg.wParam;
}
```

注意，我们不必再把窗口过程传递给类生成器了。类生成器可以使用一般性 Win::Procedure 函数，整个 Win::Procedure 函数可以由应用通用 Win::Controller 接口实现。

```
Win::ClassMaker::ClassMaker(char const * className,
                            HINSTANCE hInst)
{
  _class.lpfnWndProc = Win::Procedure;
  ...
}
```

真正区分不同窗口行为的是传递给 Win::Maker::Create 的不同控制器。图 16-1 示意了以上小库中各个组件之间的交互关系与相互依赖性。

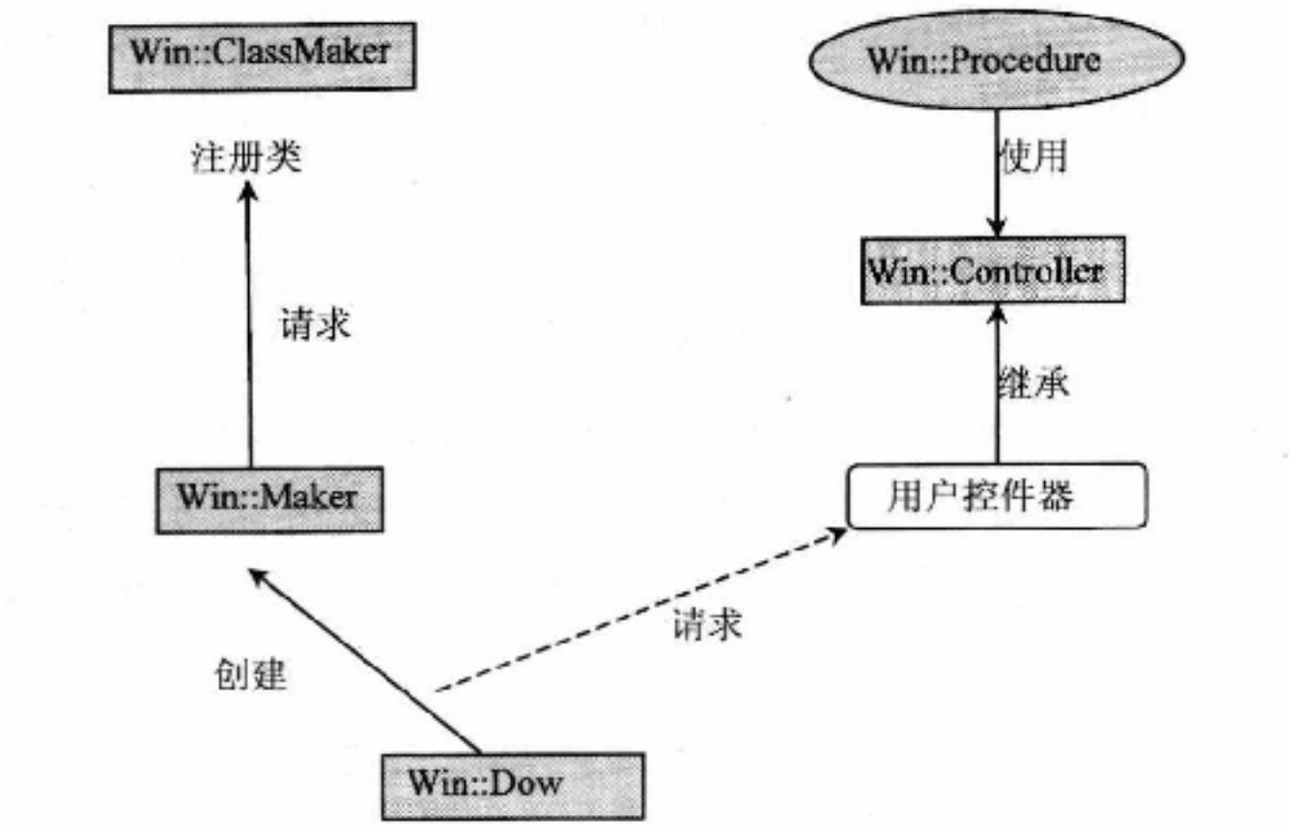


图 16-1 Windows 库中类与过程间的一些关系

这种简化的代价通常表现在程序的变大以及运行速度的变慢。

首先我们讨论速度。对每条消息来说，即使应用程序不对它进行处理，也要经过参数解包和虚方法调用这样一个过程。这是一项庞大的工作吗？我不这样认为，一个普通的窗口每秒接收到的消息并不很多。实际上，消息是存放在一个队列中，如果窗口没有处理队列中的消息，那么新消息就会覆盖旧消息。我们以鼠标移动消息来举例说明：不管你在窗口中多快地移动鼠标，你的窗口过程都不会被这些消息忙得喘不过气来。只要最后一条消息在队列中，前面的消息丢失一些是没关系的。当鼠标在鼠标垫上移动时，它发送消息的频率是相当随机的。对现在的快速处理器来说，处理一条窗口消息所用的时间是非常少的。

虽然现在的计算机的内存都很大，1 MB 的内存根本就是九牛一毛，但程序大小仍然是一个值得考虑的因素（尽管从互联网上下载程序时，1 MB 的大小并非总是可以轻松地忽略不计）。一个完整的 Win::Controller 具有与窗口消息同样多的虚方法。具体数目是多少？大约 200 个。一个完整的 vtable 大约是 800 个字节，还不足 1000 个字节!相对而言，一个简单的图标文件就有 2K 字节。在你的程序中可能会有 12 个控制器，所有的 vtable 加起来大小也不超过 10K 字节。当然每个 Win::Controller 方法也包含有缺省实现代码。代码的大小取决于编译器的优化程度，但加起来也不过几 K 字节。

最坏的情况是，假设程序里有十几种不同类型的窗口，此时情况就已经相当复杂了，而且还包含很多图标和位图。从这个角度看，为编程简单和方便付出的速度和文件大小方面的代价是最小的，是值得的。

16.2 异常定义

如果一个控制器方法发生异常会出现什么结果呢？它会越过 Win::Procedure，然后越过

若干层 Windows 代码，最终出现在消息循环中。从原理上讲，可以在 WinMain 中捕获它。捕获之后，最好的处理方法是显示一条错误提示然后退出。但不仅如此，Windows 如何对异常作出反应我们并不清楚。譬如，可能不能释放某些资源，甚至进入一种不稳定状态。最坏的情况是 Windows 根本就没有预料到窗口过程会抛出异常。这些情况都有可能导致其他程序不能运行甚至系统崩溃，所以我们必须对这种情况进行处理。

我们有两种选择：一是在 Win::Procedure 过程的 switch 语句部分加上一个 try/catch 块；其二是确保任何 Controller 方法都不会抛出任何异常。try/catch 块可能会增加每条消息的处理时间（取决于编译器），判断它是否被客户覆盖，不过，这是个小问题。但我们还是要面对如何处理异常这个问题。终止程序？这似乎并非上策!

那么不让消息处理器抛出异常怎么样？不过，这也好像也不太可能。怎么办呢？

Script 5

下面讲解异常的定义。在 C++中，可以声明函数和方法抛出何种异常。实际上，我们可以规定某种方法无抛出异常。声明的格式为：

```
virtual bool OnDestroy()throw();
```

这样的声明确保 OnDestroy()（以及派生类中的所有覆盖方法）不会抛出任何异常。异常定义的一般语法列出了过程可能抛出的异常类型：

```
void Foo()throw(bad_alloc, char *);
```

这种约束有多强呢？不幸的是，标准里只提到了编译时间。编译器只需检测在基类方法及其派生类覆盖方法之间的异常定义不匹配。异常定义尤其在派生类中约束更强（允许的异常数更少）。编译器甚至不去检测最明显的约束违背——例如，在定义为 throw()的方法内部显式抛出异常（什么也不抛出）。不过，我们希望编译器开发人员能满足程序员的要求，在违背异常定义时至少让编译器发出警告。就像让编译器报告违反 const 定义，程序员可以根据它来跟踪异常定义的违反。

异常的所有定义可以在一个标准兼容性编译器中完成，这样可以保证所有未定义异常都会转换成调用库函数 unexpected，它在缺省时终止程序的运行。这样做暂时足以应付。把 Win::Controller 的所有方法都声明为“什么都不抛出”至少可以迫使覆盖方法的客户好好想想是否允许抛出异常。

请注意，你仍然可以用异常来报告错误。不过，要小心不要让异常传递到消息处理器中，或者在消息处理器中加入 try/catch 块。

16.3　清理

Script 6

现在该把我们的库文件和应用程序文件分离开了。创建一个名字为 Lib 的子目录，把所

有的库文件都拷贝到这个目录中。作为清理工作的一部分，我们还把 TopController 的定义移到一个单独的文件 Ctrl.h 中。

> 有些编译器需要用户指定存储在其他目录下的包含文件。例如，你必须在编译器的“include”路径中添加目录名 Lib。

Script 7

第17章 绘 图

基于 Windows 环境的一个主要优点就是，程序相当容易把数据以最适合浏览和操作的形式显示出来。Windows 可以把图形和文本无缝结合起来。本章将给大家介绍 Windows 图形编程的一些基础知识。

17.1 应用程序图标



Script 8

每个 Windows 程序都必须有一个图标。当你浏览可执行文件保存所在的某个目录时，Windows 资源管理器会把可执行程序的图标显示出来。当运行可执行文件时，在任务栏上会显示该程序的图标，同时在程序窗口的左上角也会显示该图标。如果你没有为你的可执行程序提供一个图标，Windows 会使用缺省的图标。

为应用程序的标题栏指定图标的明显位置是顶层窗口的窗口类。实际上，最好是同时准备两个图标，一个大图标和一个小图标。如果只有一个图标的话，Windows 会对它进行缩放，得到的结果可能会失真。

下面我们给 Win::ClassMaker 增加一个 SetIcons 方法，并在这个类中嵌入两个图标对象。

```
namespace Win
{
  class ClassMaker
  {
  public:
    ...
    void SetIcons(int id);
  protected:
    WNDCLASSEX    _class;
```

```
    StdIcon         _stdIcon;
    SmallIcon       _smallIcon;
  };
}
```

我们很快就会给出 StdIcon 和 SmallIcon 的实现代码。先让我们来看 SetIcons 的实现代码。图标对应的图像文件一般从程序的资源中装载。

```
void Win::ClassMaker::SetIcons(int id)
{
  _stdIcon.Load(_class.hInstance, id);
  _smallIcon.Load(_class.hInstance, id);
  _class.hIcon = _stdIcon;
  _class.hIconSm = _smallIcon;
}
```

程序的资源包括图标、位图、字符串、鼠标指针、对话框模板等，它们都可以应用于可执行文件中。这样一来，当程序运行时，就不必再从磁盘中去搜索包含这些资源的文件了，直接从可执行文件中加载即可。

那么在装载资源的时候，如何识别不同的资源呢？WinMain 是通过传递给它的实例句柄来区分程序的整套资源。你可以给某资源赋予一个名字或者一个整数 ID 号。为简单（和提高效率）起见，我们将使用 ID。

让我们从基类 Win::Icon 开始。当你要加载一个图标时，你必须指定可以找到此资源的路径（实例句柄）、特定图标的唯一 ID、尺寸（以像素为单位，如果不符合实际图标尺寸，Windows 会对它进行位伸或缩小）以及一些随机标志。

```
namespace Win
{
  class Icon
  {
  public:
    Icon(HINSTANCE res,
        int id,
        int dx = 0,
        int dy = 0,
        unsigned flag = LR_DEFAULTCOLOR)
    {
      Load(res, id, dx, dy, flag);
    }
    ~Icon();
    operator HICON()const { return _h; }
protected:
    Icon(): _h(0){}
    void Load(HINSTANCE res,
              int id,
              int dx = 0,
              int dy = 0,
```

```
                unsigned flag = LR_DEFAULTCOLOR);
  protected:
    HICON _h;
  };
}
```

编译器使用转换运算符 HICON 可以把一个 Icon 类型的对象（或者这个对象的引用）直接隐式转换成 Windows 定义的 HICON 类型。当你想用一个 Icon 对象调用参数类型为 HICON 的 Windows API 函数，但你希望传递一个 Icon 对象时，这一途径很方便。虽然这两者并没有继承与被继承的关系（HICON 甚至不是一个类或者结构），你甚至可以说 Icon 就是 HICON。

装载图标的 API 函数是 LoadImage，它还可以用来装载其他类型的图像。它的返回类型不确定，所以必须转换成 HICON 类型。一旦图标不再使用，应该调用 DestroyIcon 销毁它。

```
void Icon::Load(HINSTANCE res, int id,
                int dx, int dy, unsigned flag)
{
  _h = reinterpret_cast<HICON>(
      ::LoadImage(res,
                  MAKEINTRESOURCE(id),
                  IMAGE_ICON,
          dx, dy,
          flag));
  if(_h == 0)
     throw "Icon load image failed";
}

Icon::~Icon()
{
  ::DestroyIcon(_h);
}
```

注意，我们不能直接把图标 ID 传递给 API 函数，必须使用 MAKEINTRESOURCE 宏进行间接性的转换。

> LoadImage 和其他几个与资源有关的 API 函数并不知道传递过来的参数是字符串形式的文件名还是整数形式的 ID。而所有的 API 函数都是 C 语言函数，不能被重载。所以必须应用一定的技巧使这些函数能够接受这两种类型的传递值并让它们猜测其真正身份。MAKEINTRESOURCE 就是这样的一个宏，它对整数的位进行操作使它看起来和字符指针不一样（在 Windows API 函数刚问世的那个时期，这种编程方法非常流行）。

下面我们马上可以得到 Icon 的子类：SmallIcon 和 StdIcon。它们的构造函数和 Load 方法更简单——它们不需要图标的尺寸和标志。

```
namespace Win
{
  class SmallIcon: public Icon
```

```
{
public:
  SmallIcon(){}
  SmallIcon(HINSTANCE res, int id);
  void Load(HINSTANCE res, int id);
};

class StdIcon: public Icon
{
 public:
   StdIcon(){}
   StdIcon(HINSTANCE res, int id);
   void Load(HINSTANCE res, int id);
 };
}
```

Load 方法由父类的 Icon::Load 方法实现（这里必须加上父类名和两个冒号以消除歧义，否则编译器会认为这是一个递归调用，程序会进入死循环）。

为了得知标准图标和小图标的正确尺寸，我们调用了通用的 API 函数 GetSystemMetrics，根据此函数可以获得许多当前系统的缺省值。

```
void SmallIcon::Load(HINSTANCE res, int id)
{
  Icon::Load(res, id,
        ::GetSystemMetrics(SM_CXSMICON),
        ::GetSystemMetrics(SM_CYSMICON));
}

void StdIcon::Load(HINSTANCE res, int id)
{
  Icon::Load(res, id,
        ::GetSystemMetrics(SM_CXICON),
        ::GetSystemMetrics(SM_CYICON));
}
```

还有一个问题，我们如何创建图标？有两条路，一条难的，一条容易的。难的方法就是应用某种图标编辑器，自己编写命名图标文件的资源脚本，然后，用特定的工具对它进行编译并把它链接到可执行文件中。

为了让你有一个更清楚的印象，下面我们描述一下具体细节。假设你的资源脚本文件叫做 script.rc，它必须包含下面两行：

```
#include "Resource.h"
IDI_MAIN   ICON   "main.ico"
```

IDI_MAIN 是在 Resource.h 中定义的一个常量。关键字 ICON 表明它对应于一个图标。后面的“main.ico”则是图标文件的名字。

头文件 Resource.h 中包含常量的定义，例如：

```
#define IDI_MAIN  101
```

其中的数字 101 是任意指定的。

不过，不能使用更安全的 C++版本：

```
const int IDI_MAIN = 101;
```

脚本文件编写完之后，必须使用程序 rc.exe（资源编辑器）对它进行编译，得到 script.res 文件。链接器会把 script.res 文件和其他对象文件链接到一个可执行文件中。

创建图标的较容易办法是使用 IDE（integrated development environment，集成开发环境）以及资源编辑器。你可以在资源编辑器中创建图标，添加到资源中并给它指定一个相应的符号 ID，让集成环境替你完成剩下的工作（在对话框的设计中，图形资源编辑器是必不可少的）。

注意在上例中，两个图标使用了同一个 ID 号。这种情况是允许的，因为同一个图标文件可以包含两个（或更多）不同尺寸大小的图像。调用 LoadImage 时，它会从图标文件中选取一个尺寸最接近的图像。一般来说，至少要创建 32×32 和 16×16 两个图标（单位为像素）。

我已经创建了两个图标，并给它们指定了整数 ID：IDI_MAIN（在 Resource.h 中定义，在 main.cpp 中必须包含 Resource.h）。现在剩下的就是在 WinMain 中增加一次调用。

```
Win::ClassMaker winClass(className, hInst);
winClass.SetIcons(IDI_MAIN);
winClass.Register();
```

程序资源中的图标创建好之后，Windows 命令解释器会找到它并使用它进行显示。如果资源中有多个图标，Windows 会选择 ID 号最小的那个图标。

17.2　窗口绘图和视图对象

和 Windows 中的其他操作一样，窗口绘图是对外界动作的响应。例如，程序可以随着鼠标的移动或者点击画出某些图形；随着键盘的敲击显示相应的字符等。程序用来绘图的那部分窗口一般叫做客户区，它不包括窗口的边界、标题栏、菜单等——这些是非客户区。

还有一种情况，Windows 会要求应用程序重绘部分或者全部客户区。这种情况的发生也许是由于 Windows 的惰性（或者资源缺乏）。如果你的窗口被别的程序窗口覆盖了，系统会舍弃覆盖图像部分。当你的窗口去掉了覆盖窗口后，必须重绘舍弃的图像部分。那么这个工作由谁做呢？由你的程序。当你把窗口最小化，然后又最大化，或者改变窗口大小，都会出现这种情况。

从应用的角度来看，这些动作或多或少是随机发生的，程序必须随时做好准备完全重绘整个客户区。Windows 有一个与之相关的专用消息 WM_PAINT，当 Windows 需要用户程序协助重绘窗口时，它会向用户程序发送 WM_PAINT 消息。另外，当窗口第一次显示时，Windows 也会向程序发送这条消息。



Script 9

为了演示绘图的原理，下面把我们的 Windows 程序进行一下扩展，让它可以跟踪鼠标移动。当鼠标移动时，程序将在鼠标光标的新位置和旧位置之间画一条线。不过在这之前，我们想在程序中添加模型—视图—控制器三对象组中的第二个对象。视图将负责所有的绘图操作，它还将存储最后记录下来的鼠标光标位置。

```
class TopController: public Win::Controller
{
  ...
private:
  View _view;
};
```

17.3 画布

所有的显示操作都必须要输出到某个设备上，比如屏幕、打印机、绘图仪或其他一些设备。在一个窗口中绘图，程序必须要为这个窗口客户区获得客户与非客户区的设备环境（DC，device context）。Windows 可以在内部创建一个 DC，然后把它的句柄交给我们。然后我们在所有的窗口输出操作中使用这个句柄。输出完成之后，必须释放这个句柄。

DC 是一种资源，对它最好的处理方法是应用资源管理方法。我们将把 DC 一般性拥有者叫做 Canvas（画布）。根据 DC 的创建和销毁方法，画布分很多类型，但是所有画布的功能都是一样的。例如，我们可以调用任一个 Canvas 对象画一条线或者打印一些文本。下面我们以这两种操作作为实现绘图功能的开始。

```
namespace Win
{
  class Canvas
  {
  public:
    operator HDC()
    {
      return _hdc;
    }
  void Line(int x1, int y1, int x2, int y2)
  {
    ::MoveToEx(_hdc, x1, y1, 0);
    ::LineTo(_hdc, x2, y2);
  }
  void Text(int x, int y, char const * buf, int count)
  {
    ::TextOut(_hdc, x, y, buf, count);
  }
```

```
  protected:
    Canvas(HDC hdc):_hdc(hdc){}

    HDC  _hdc;
  };
}
```

HDC（设备环境的句柄）是一个 Windows 数据结构，DC 的一个句柄。

一般性类 Canvas 没有提供对此句柄进行初始化的公共方法——必须由子类对它初始化。成员运算符 HDC 的功能是把 Canvas 隐式转换成 HDC。当给一个要求 HDC 的 API 传递 Canvas 对象时，这一运算符就会发挥作用（同样可以把它看作“Canvas 是一个 HDC”的关系的表达）。

为了在两点之间画一条线，我们要调用两个 API。第一个是 MoveToEx，它的功能是设置“当前位置”；第二个是 LineTo，它的功能是从当前位置到参数指定点之间画一条线（同时当前位置也移到指定点）。点的位置包括 x 和 y 两个坐标。在缺省的坐标系中，它们的单位是屏幕像素。原点处的坐标为 x=0、y=0，位于窗口客户区的左上角。x 坐标从左到右递增；y 坐标从上到下递增。

要打印文本，就必须指定它在窗口中显示的位置。传递给 TextOut 的 x，y 坐标是告诉 Windows 放置字符串左上角的位置。这和输出到标准输出设备有所不同，在标准输出设备上，位置的惟一控制方法是换行符。对于 Windows DC 来说，换行符是没有意义的（和其他不可打印字符一样）。事实上，终止字符串的空（null）字符对 Windows 来说也是没有意义的。用 TextOut 进行输出的字符串不需要用空字符结束。相反，你只需指定要打印的字符个数。

那么我们如何以及从何处得到 DC 呢？既然我们想在每一次鼠标移动时都画一条线，所以我们要在 WM_MOUSEMOVE 消息的处理器中完成。这就意味着，我们的控制器要覆盖 Win::Controller 的虚方法 OnMouseMove。

从 Windows 的 WM_PAINT 消息处理过程之外获得的 DC 的画布类型叫做 UpdateCanvas。获取和释放 DC 的一对 API 函数分别是 GetDC 和 ReleaseDC。

```
namespace Win
{
  class UpdateCanvas: public Canvas
  {
public:
  UpdateCanvas(HWND hwnd)
      :  Canvas(::GetDC(hwnd)),
         _hwnd(hwnd)
  {}
  ~UpdateCanvas()
  {
    ::ReleaseDC(_hwnd, _hdc);
  }
protected:
```

```
    HWND _hwnd;
  };
}
```

在适当的 Controller 方法中创建 Canvas 或者 Canvas 的一个派生对象——本例中，我们在 OnMouseMove 方法中创建。这样，View 方法能独立于传递过来的实际 Canvas 类型而工作。

```
bool TopController::OnMouseMove
  (int x, int y, Win::KeyState kState)throw()
{
  Win::UpdateCanvas canvas(_h);
  _view.MoveTo(canvas, x, y);
  return true;
}
```

下面我们准备实现 View 对象。

```
class View
{
public:
  View(): _x(0), _y(0){}
  void MoveTo(Win::Canvas & canvas, int x, int y)
  {
    canvas.Line(_x, _y, x, y);
    _x = x;
    _y = y;
    PrintPos(canvas);
  }
private:
  void PrintPos(Win::Canvas & canvas)
  {
    std::string str("Mouse at: ");
    str += ToString(_x);
    str += ", ";
    str += ToString(_y);
    canvas.Text(0, 0, &str [0], str.length());
  }
private:
  int _x, _y;
};
```

PrintPos 方法很有趣。它的功能是打印：“Mouse at:”，再加上鼠标位置的 x，y 坐标，例如“Mouse at: 124, 211.”。我们希望这个字符串出现在客户区的最左上角，即坐标（0,0）处。

首先，我们必须格式化字符串。特别是要把两个数字转换成字符串形式。数字的打印格式内建在标准流中，所以我们只要使用基于字符串的流。实际上，任何能被流接受的类型都可以用下面这个简单的模板函数转换成一个字符串：

```
#include <sstream>
template<class T>
std::string ToString(T const & val)
{
  std::ostringstream out;
  out << val;
  return out.str();
}
```

我们使用运算符+=把不同的字符串联起来。最后，通过取出字符串中第一个字符的地址将字符串转换成字符指针，这也正是 API 函数的要求。

下面可以运行这个简单的程序了。将鼠标指针移到客户区之上，随着鼠标的移动，你会看到鼠标在屏幕上留下的轨迹，窗口左上角还会显示鼠标的坐标。不过，你同时会发现这个程序的几个不足之处。例如，你把窗口最小化，然后最大化，所有的轨迹都消失了，坐标指示器也没有了。另外，如果别的应用程序窗口覆盖了此程序的部分窗口，然后再把覆盖窗口移走，被覆盖部分变成了空白区域——鼠标轨迹被删除了。改变窗口大小时，情况也是这样。

还有一些其他的小问题。例如，当鼠标离开窗口后，它的位置没有更新，等鼠标重新进入窗口客户区范围，程序会在最后存储的鼠标位置和新进入点之间画一条线，而这条线并不是实际的轨迹。

为了解决以上问题，如果把鼠标从窗口的右下角移出，从窗口的左上角移入。这时，显示鼠标坐标的字符串变短了（位数变少了），但前一个字符串的尾数并没有删除。

下面我们一个一个来解决上面的这些问题。

17.4　WM_PAINT 消息

Script 10

首先，每个优秀的 Windows 应用程序必须能对 WM_PAINT 消息作出响应。而我们上面的那个程序却没有对 WM_PAINT 消息作出响应，所以当窗口被覆盖、最小化后再恢复、改变大小时，客户区没有被重绘。那么当 Windows 要求我们恢复客户区时，我们应该做什么呢？显然，最好的解决方案就是重绘鼠标轨迹和重新显示鼠标坐标。但困难在于我们不记得鼠标的轨迹。那么，我们就先从简单的修改开始——重新显示坐标。

在窗口过程中加入一个新的 case 块：

```
case WM_PAINT:
  if(pCtrl->OnPaint())
      return 0;
  break;
```

严格来说，WM_PAINT 有一个参数 WPARAM，在某些特殊情况下 WPARAM 与常规控件有关，它可以被设置为 DC。现在，我们暂时不管这个参数，把注意力集中于一般情况。

我们还要更新我们的 Windows 库，给通用的 WinController 添加一个哑元虚方法 OnPaint。

```
virtual bool OnPaint()throw(){ return false; }
```

获取响应 WM_PAINT 消息 DC 的标准办法是调用 API BeginPaint 函数。处理完之后，必须调用匹配的 EndPaint API 函数释放此 DC。所有权功能性被很好地封装在 PaintCanvas 对象中：

```
namespace Win
{
  class PaintCanvas: public Canvas
  {
public:
  PaintCanvas(HWND hwnd)
    : Canvas(::BeginPaint(hwnd, &_paint)),
      _hwnd(hwnd)
  {}
  ~PaintCanvas()
  {
    ::EndPaint(_hwnd, &_paint);
  }
  int Top()const    { return _paint.rcPaint.top; }
  int Bottom()const { return _paint.rcPaint.bottom; }
  int Left()const   { return _paint.rcPaint.left; }
  int Right()const  { return _paint.rcPaint.right; }
protected:
    PAINTSTRUCT _paint;
    HWND        _hwnd;
  };
}
```

请注意，BeginPaint 通过填充 PAINTSTRUCT 结构，允许调用者访问一些额外有用信息。尤其是可以得到需要重绘矩形区域的坐标。在很多情况下，需要重绘的区域只是客户区的一小部分（例如，移走小部分覆盖窗口或微量调整窗口的大小后）。在我们这个并不复杂的应用程序中，我们不使用这些额外的信息——我们把整个窗口彻底重绘一遍。

下面是我们对控制器 OnPaint 方法的覆盖方法。它创建了一个 PaintCanvas 并调用相应的 View 方法。

```
bool TopController::OnPaint()throw()
{
  Win::PaintCanvas canvas(_h);
  _view.Paint(canvas);
  return true;
}
```

View 调用它的私有方法 PrintPos。注意 View 不区分 UpdateCanvas 和 PaintCanvas。它所知道的就是，给它的是一个一般性 Win::Canvas。

```
void View::Paint(Win::Canvas & canvas)
{
  PrintPos(canvas);
```

```
}
```

对于不同大小的打印字符串来说，我们该如何处理呢？我们需要进行更多的格式控制。在下面的代码中，我们把 std::setw（4）操纵器传递给输出流，确保两个坐标都以固定的 4 位宽度显示。如果下一个要打印的数少于四位数，程序会自动给它补上空格。setw 和其他流操纵器在<iomanip>中定义。

```
void View::PrintPos(Win::Canvas & canvas)
{
  std::stringstream out;
  out << "Mouse at: " << std::setw(4)<< _x;
  out << ", " << std::setw(4)<< _y;
  out << "    ";
  std::string s = out.str();
  canvas.Text(0, 0, &s [0], s.length());
}
```

你也许会注意到，根据防错性编程原则，我在字符串后面加了四个空格。实际上，如果没有这些空格的话，我们还将面临字符串缩短的问题。现在我们把字符串中字符的个数固定为 24，这也不能保证字符串总是显示在同一区域。这是因为 Windows 使用的是比例字体，有些字符比其他字符要宽。特别是空格符比非空格字符要窄。

当用户改变窗口大小时，WM_PAINT 消息也会被发送到程序。改变窗口大小也是只需重绘新增加区域的情况之一。新矩形的坐标可以通过 PaintCanvas 获得。

不过实际上并不像听起来这么简单。例如在某些应用程序中，任何窗口大小的改变都会导致整个窗口的重绘。请想象一下，想一下那种把图片拉伸填满整个客户区的窗口，改变窗口大小就需重绘整个窗口的复杂性。

即使对这样的应用程序，除了窗口大小改变之外，还可以把重绘的区域限制在那些被刷新覆盖的部分。那么，我们如何才能如愿以偿呢？实现这一点的关键在注册窗口类时设置两个样式位。这两个位分别是 CS_HREDRAW 和 CS_VREDRAW，前者告诉 Windows 在窗口的水平大小（宽度）改变时要求全部重绘（也就是刷新整个客户区域）。后者的功能一样，不过是针对窗口高度改变的情形。你可以通过二元位或（or）操作组合设置它们：CS_HREDRAW | CS_VREDRAW。

刷新一个区域不仅会向窗口发送 WM_PAINT 消息及对应的边界矩形，还将使用背景刷把区域重新绘制一遍。背景刷在窗口类定义中设置——缺省设置为 COLOR_WINDOW 标准刷（如果背景刷被设为 0，则 Windows 不会擦除区域中的内容——试一试）。

17.5　模型

Script 11

如果我们要进行有意义的重绘，那么我们必须存储鼠标移动的历史轨迹。为此，我们就

要利用“模型—视图—控制器”中的模型部分。通常，你都不想让视图或控制器具有太多的智能性，或记住太多的历史数据。鼠标轨迹就是我们程序的数据，这是与 I/O 无关的部分。你甚至可以想像成一个命令行程序版本。它无法可视化地显示线条，并且你还必须输入每个点的坐标来确定对应点——不过在模型中也是这样的。

我决定使用一个新的数据结构——C++标准库中的双端队列（deque）（读作“deck”）。它的工作原理像一个双端矢量。在它的两端都可以进行推入和弹出操作，而且 push_front 和 pop_front 与 push_back 和 pop_back 的工作效率一样。

我们不希望历史越过 MAX_SIZE 临界点。所以，当我们加入的新点导致双端队列超过 MAX_SIZE 时，就把最早的点弹出。事实上，这种机制与传统队列或者 LIFO（后入先出）数据结构吻合。

```
#include <deque>
#include <utility> // Pair

class Model
{
  enum { MAX_SIZE = 200 };
public:
  typedef
  std::deque< std::pair<int, int> >::const_iterator iter;

  Model()
  {
    AddPoint(0, 0);
  }
  void AddPoint(int x, int y)
  {
    if(_queue.size()>= MAX_SIZE)
      _queue.pop_front();
    _queue.push_back(std::make_pair(x, y));
  }
  iter begin()const { return _queue.begin(); }
  iter end()const { return _queue.end(); }
private:
  std::deque< std::pair<int, int> > _queue;
};
```

从上面的程序我们可以看到，轨迹点以整数 std::pairs 形式存储。为一个二维点创建一个新的数据结构太麻烦了。如果你不想显式指定成员对的类型时，可以使用方便的 make_pair 函数。让编译器根据参数类型（此处的两个参数都是整数）推断其类型。

```
_queue.push_back(std::pair<short, short>(x, y));
```

控制器必须可以访问模型和视图。当鼠标移动时，控制器给模型增加一个新轨迹点，并且像以前一样，通知视图移到新位置。

```
bool TopController::OnMouseMove
 (int x, int y, Win::KeyState kState)throw()
{
  _model.AddPoint(x, y);
  Win::UpdateCanvas canvas(_h);
  _view.MoveTo(canvas, x, y);
  return true;
}
```

现在，程序可以更智能地重绘窗口了（虽然比较浪费，但我们还是重绘了整个客户区，而不仅仅是 PaintCanvas 传递过来的矩形区）。我们从模型获得迭代器，并把它传给视图。迭代器允许视图访问被记忆的历史轨迹部分。

```
bool TopController::OnPaint()throw()
{
  Win::PaintCanvas canvas(_h);
  _view.Paint(canvas, _model.begin(), _model.end());
  return true;
}
```

现在视图所要做的是把所有的点连起来。我们使用了 C++标准库<algorithm>部分的一个小技巧。for_each 算法包含启动迭代器、结束迭代器以及函数符。

我们已经见过在排序算法中利用函数符作为断言的例子。在这里我们将使用这样一种函数符，它可以操作迭代器“指向”的对象。在我们的程序中，迭代器“指向”坐标对——也就是说，当它被反引用后，它返回一个指向 std::pair<int, int>的引用。

我们的函数符叫做 DrawLine，它在最后记忆的位置与坐标对传递过来的位置之间画一条线。我们必须用起始位置初始化函数符，对迭代器送来的每一个值均让 std::for_each 模板调用一次此算符。

```
void View::Paint(Win::Canvas & canvas,
                 Model::iter beg,
                 Model::iter end)
{
  PrintPos(canvas);
  if(beg != end)
  {
    DrawLine draw(canvas, *beg);
    ++beg;
    std::for_each(beg, end, draw);
  }
}
```

不必好奇 for_each 是如何实现的，其实很简单，请看下面的代码：

```
template<class Iter, class Op>
inline Op for_each(Iter it, Iter end, Op operation)
{
  for(; it != end; ++it)
```

```
    operation(*it);
  return(operation);
}
```

这里的 Op 可以是一个全局函数的指针，或者重载了函数调用运算符的某个类的对象。下面是函数符的实现代码：

```
class DrawLine
{
public:
  DrawLine(Win::Canvas & canvas,
           std::pair<int, int> const & p)
    : _canvas(canvas)
  {
    _canvas.MoveTo(p.first, p.second);
  }
  void operator()(std::pair<int, int> const & p)
  {
    _canvas.LineTo(p.first, p.second);
  }
private:
  Win::Canvas & _canvas;
};
```

Canvas 的两个新方法 MoveTo 和 LineTo 的实现都很简单：

```
void Canvas::LineTo(int x, int y)
{
  ::LineTo(_hdc, x, y);
}
void Canvas::MoveTo(int x, int y)
{
  ::MoveToEx(_hdc, x, y, 0);
}
```

17.6 捕获鼠标

其实，无条件地绘出鼠标轨迹没有多大的意义。如果你用过其他 Windows 图形编辑器，你就会知道在画线时一般都要按下鼠标按钮，否则你根本没有机会“提起笔来”访问其他控件或到另外一个地方继续绘画。

从原理上，我们可以使用 Win::KeyState 的参数来捕获 OnMouseMove，检查鼠标按钮是否被按下。但我们可以在此基础上做得更好，在检查按钮是否被按下的同时还能解决另外一个问题，也就是跟踪移到窗口之外的鼠标。

通常鼠标消息只发送给它所在的那个窗口。但是如果你的窗口“捕获”了鼠标，Windows 会把所有的鼠标消息重定向到此窗口——即使鼠标不在此窗口中。显然捕获鼠标要小心行事。一旦鼠标被捕获，用户就不能再点击任何控件，也不能和其他应用程序交互（如激活它）。所

以习惯做法是在鼠标拖动时捕获它——而拖动是从捕捉窗口内开始的（拖动是指按下鼠标按钮移动鼠标）。

与鼠标捕获有关的三个 API 函数是：SetCapture、ReleaseCapture 和 GetCapture。SetCapture 获取想捕捉鼠标的窗口的句柄；ReleaseCapture 结束捕获过程，并向捕获窗口发送 WM_CAPTURECHANGED 消息，GetCapture 返回拥有当前捕获的应用程序窗口的句柄（严格来说是当前线程中）。

Script 12

我们将把这些 API 函数添加到 Win::Dow 类中，Win::Dow 代表一个窗口句柄（由于 operator HWND（）的存在，Win::Dow 可当成一个 HWND）。

```
namespace Win
{
  class Dow
  {
  public:
    Dow(HWND h = 0): _h(h){}
    void Init(HWND h){ _h = h; }
    operator HWND()const { return _h; }
    template <class T>
inline T GetLong(int which = GWL_USERDATA)
{
  return reinterpret_cast<T>(
                  ::GetWindowLong(_h, which));
}
template <class T>
inline void SetLong(T value, int which = GWL_USERDATA)
{
  ::SetWindowLong(_h, which,
                  reinterpret_cast<long>(value));
}
void CaptureMouse()
{
  ::SetCapture(_h);
}
void ReleaseMouse()
{
  if(HasCapture())
    ::ReleaseCapture();
}
bool HasCapture()const
{
  return ::GetCapture()== _h;
}
```

```
// 窗口可见度
void Show(int cmdShow = SW_SHOW)
{
  ::ShowWindow(_h, cmdShow);
}
void Hide()
{
  ::ShowWindow(_h, SW_HIDE);
}
    void Update()
    {
      ::UpdateWindow(_h);
    }
    void Display(int cmdShow)
    {
      Show(cmdShow);
      Update();
    }
  private:
    HWND _h;
  };
}
```

典型的鼠标捕获过程如下：

1. 当用户按下相应的鼠标按钮时捕获鼠标；
2. 在处理鼠标移动消息时，检查是否拥有捕获。如果有，则实现拖动行为；
3. 当用户放开鼠标按钮时，释放鼠标捕获；
4. 当收到 WM_CAPTURECHANGED 消息时，结束拖动。

另外还有很重要的一点就是要在 WM_CAPTURECHANGED 消息处理器中结束拖动，而不是在按钮松开时直接作出反应。因为窗口有可能因为其他一些原因而丢失捕获，在这种情况下清理工作也是很重要的（例如，系统显示警告信息时，外部初始捕获发生改变）。

我们通过覆盖 WM_LBUTTONDOWN、WM_LBUTTONUP 和 WM_CAPTUREC- HANGED 的处理器来应用这一过程。Windows 库中的 Win::Controller 要做相应的改变，包括用 Win::Dow 替换 HWND，以使得窗口操作更简单。

```
class Controller
{
public:
  ...
  virtual bool OnLButtonDown
        (int x, int y, Win::KeyState kState)throw()
    { return false; }
  virtual bool OnLButtonUp
        (int x, int y, Win::KeyState kState)throw()
```

```
    { return false; }
  virtual bool OnCaptureChanged
        (HWND hwndNewCapture)throw()
    { return false; }
protected:
  Win::Dow  _h;
};
```

下面是客户自定义 TopController 的新方法的实现（注意在这之后，我们要删除 OnCaptureChange 的实现代码，因为我们没有必要再对拖动作任何总结了）。

```
bool TopController::OnLButtonDown
 (int x, int y, Win::KeyState kState)throw()
{
  _h.CaptureMouse();
  _model.AddPoint(x, y, true);              // 起始点
  Win::UpdateCanvas canvas(_h);
  _view.MoveTo(canvas, x, y, false);        // 不绘制
  return true;
}

bool TopController::OnLButtonUp
 (int x, int y, Win::KeyState kState)throw()
{
  // ReleaseMouse 将发送 WM_CAPTURECHANGED 消息
  _h.ReleaseMouse();
  return true;
}

bool TopController::OnCaptureChanged
 (HWND hwndNewCapture)throw()
{
  return true;
}
```

OnLButtonDown（鼠标左键按下）的实现有一些令人感兴趣的地方。因为现在用户可以“抬起画笔”了，所以我们可以绘出（以及重绘）一些不连续直线了。视图的 MoveTo 方法必须能够在不绘直线的情况下把鼠标移动到当前位置，另外模型必须在一条新线的起始处作一个记号。OnLButtonDown 中的两个布尔型标志就是用来完成此目的的。以后，我们还会回过头来讨论这个问题。下面我们先来看看 OnMouseMove 方法的新实现代码。

```
bool TopController::OnMouseMove
   (int x, int y, Win::KeyState kState)throw()
{
  Win::UpdateCanvas canvas(_h);
  if(_h.HasCapture())
  {
    _model.AddPoint(x, y);
```

```
      _view.MoveTo(canvas, x, y);
    }
    else
      _view.PrintPos(canvas, x, y);
    return true;
  }
```

请注意，只有当我们捕获了鼠标时才记录下鼠标的位置；否则，我们只更新鼠标坐标的显示（而且，因为我们没有捕获鼠标，所以当鼠标离开窗口边界时，我们将停止记录其坐标）。

重绘窗口稍微复杂一点，因为窗口中可能有多条不连续直线。我给每一个记忆点分配一个布尔型的标志，如果这个点是一条新线的起点（参见上面的 OnLButtonDown 实现），那么设置其标志为 true。现在我们不是用 std::pair 这样的简单形式来表示一个点。事实上，为了节省空间，我决定把坐标保存为 short 类型（显然如果屏幕分辨率突破了每行 32000 像素，我们的程序需要重写）。

```
class Model
{
  enum { MAX_SIZE = 1000 };
public:
  class Point
  {
public:
Point(int x = 0, int y = 0, bool isStart = false)
      : _x(static_cast<short>(x)),
        _y(static_cast<short>(y)),
        _isStart(isStart)
  {}
  int X()const { return _x; }
  int Y()const { return _y; }
  bool IsStart()const { return _isStart; }
private:
  short _x;
  short _y;
  bool  _isStart;
};

typedef std::deque<Point>::const_iterator iter;

void AddPoint(int x, int y, bool isStart = false)
{
  if(_queue.size()>= MAX_SIZE)
    _queue.pop_front();
  _queue.push_back(Point(x, y, isStart));
```

```
    }
    iter begin()const { return _queue.begin(); }
    iter end()const { return _queue.end(); }
  private:
    std::deque<Point> _queue;
  };
```

注意 Point 构造函数所有参数的缺省值。显然，如果我们想把 Point 存储在一个双端队列中，它就需要使用缺省（无参数）构造函数。当双端队列在内部增长时，它应该能够分配和初始化新的 Point 块。它通过调用缺省的构造函数实现这一点。

View::Paint 的实现除了 PrintPos 包含要显示的坐标值（还有更新记忆点的位置）之外没有什么改变。

```
void View::Paint(Win::Canvas & canvas,
                 Model::iter beg,
                 Model::iter end)
{
  PrintPos(canvas, _x, _y);
  if(beg != end)
  {
    DrawLine draw(canvas, *beg);
    ++beg;
    std::for_each(beg, end, draw);
  }
}
```

函数符 DrawLine 也要做相应的修改：现在鼠标只是移到当前新位置上，不再从线段的起始点画线了。

```
class DrawLine
{
public:
  DrawLine(Win::Canvas & canvas, Model::Point const & p)
    : _canvas(canvas)
  {
    _canvas.MoveTo(p.X(), p.Y());
  }
  void operator()(Model::Point const & p)
  {
    if(!p.IsStart())
      _canvas.LineTo(p.X(), p.Y());
    else
      _canvas.MoveTo(p.X(), p.Y());
  }
private:
  Win::Canvas & _canvas;
};
```

17.7 添加颜色和修饰

Script 13

下面来给我们的程序增加一点乐趣吧。比如，如何画出彩色线条？LineTo API 不包含指定颜色的参数。

封装 Canvas 对象中的设备环境（DC）保存了所有用于画图的参数。我们必须引导画布改变其绘图颜色。就像一个文明的“市民”一样，在画完之后，我们还要把颜色改回来。

不同的画图和打印模式可以方便地在 GDI（graphics device interface，图形设备接口）对象中编制成组。画笔用于控制绘制直线。你可以选择某个对象来改变 DC 的属性。同样，如果要做个好“市民”，在完成操作之后，通常要取消选择当前对象，并恢复原来的对象。

这种对象选择机制可以理想地封装于不同资源管理对象中。下面，我们从最普通的对象类型（库存对象）开始。Windows 以预定义对象的形式提供了一整套最常用的设备环境设置。等一会我们就会看到 Windows 的“库存”是多么丰富。现在，我们需要设计一个封装这些对象的理想封装器。

```
namespace Win
{
  class StockObject
  {
  public:
    StockObject(int type)
      : _obj(::GetStockObject(type))
    {}
    operator HGDIOBJ()const { return _obj; }
  protected:
    HGDIOBJ _obj;
  };

  class StockObjectHolder
  {
  public:
    StockObjectHolder(HDC hdc, int type)
      : _hdc(hdc)
    {
      _hObjOld = ::SelectObject(_hdc, StockObject(type));
    }

    ~StockObjectHolder()
    {
      ::SelectObject(_hdc, _hObjOld);
```

```
    }
  private:
    HGDIOBJ  _hObjOld;
    HDC      _hdc;
  };
}
```

Win::StockObjectHolder 的构造函数带有设备环境句柄（也就是说，我们可以给它传递给任何 Canvas 对象），并选择一个特定的库存对象。SelectObject API 返回前这一类型的前一个对象，我们把这个对象保存下来，以备后用。另外我们还保存了 DC，所以在析构函数中，我们可以把前一个对象恢复到 DC 中的正确位置。

上面都是资源管理部分的内容。不同类型的库存对象由预定义的整数 ID 区分。有了这个 ID，我们可以用 GetStockObject 从 Windows 的储藏室中取出正确的对象。在大部分情况下，我们都使用 Win::StockObjectHolder 作为特定库存对象保存类的基类。

有几种库存画笔也许有用，例如黑色画笔和白色画笔。我们将把这些库存画笔都包含在一个新的名字空间 Pen 中，作为更一般的 Pen::Holder 类的嵌入类。现在这个类暂时是空的——不过，我们很快就会充实它。

这种类型的类嵌入使得命名更加方便。我们不仅可以指定 Pen::Holder::White 和 Pen::Holder::Black 这样的类，而且还可以指定更一般的类 Pen::Holder。

```
namespace Pen
{
  class Holder
  {
  public:
    class White : public Win::StockObjectHolder
    {
    public:
      White(HDC hdc)
        : Win::StockObjectHolder(hdc, WHITE_PEN)
      {}
    };

    class Black : public Win::StockObjectHolder
    {
    public:
      Black(HDC hdc)
        : Win::StockObjectHolder(hdc, BLACK_PEN)
      {}
    };
  }
}
```

为简单起见，我没有把名字空间 Pen 嵌套于 Win 中，所以我们不必使用 Win::Pen::Holder::Black，只需使用简写形式 Pen::Holder::Black。如果要设计一个商业库，我可能会嵌

套名字空间以避免名字冲突。

使用一个库存画笔的方法是在一给定作用域内定义一个持有者。例如，使用下面的代码来画一条白色直线：

```
{
  Pen::Holder::White wp(canvas);
  // 绘直线
}
```

别忘了给画笔持有者取个名字（在本例中为 wp）。下面的代码虽然没有产生预期的效果，但它也是有效的：

```
{
  Pen::Holder::White(canvas);
  //绘直线
}
```

编译器会创建一个匿名临时对象并立即销毁它，而不是等到作用域的最后（临时对象生存期限制于当前语句的有效期）。

如果你想画出除了黑和白之外的更多颜色（真彩），就必须创建自己的 GDI 对象。调用 CreatePen 创建一个彩色画笔，调用 DeleteObject 销毁它。通过指定三种颜色成份来定义某种颜色：红、绿、蓝。它们的取值范围为 0～255。不过，这不意味你的计算机就可以显示 256×256×256 种不同的颜色。除非你把显示模式至少设置为 24 位颜色（按 Windows 的说法是“真彩”），系统就会试着找一种最接近的颜色匹配。Windows 中有一个叫做 RGB 的宏，它可以把三个颜色成份组合到一个数组结构 COLORREF 中。

```
namespace Pen
{
  class Color
  {
  public:
    Color(int r, int g, int b, int style = PS_SOLID)
    {
      _hPen = ::CreatePen(style, 0, RGB(r, g, b));
    }
    ~Color()
    {
      ::DeleteObject(_hPen);
    }
    operator HPEN(){ return _hPen; }
  private:
    HPEN    _hPen;
  };
}
```

创建一个彩色画笔本身并不改变画线的颜色。你还必须把彩色画笔选入你的设备环境中。根据资源管理的原则，我们把这个操作封装到 Pen::Holder 对象中——与 White 和 Black

嵌入的类为同一个类。定义一个 Pen::Holder 类型对象会暂时改变接下来所有画线的颜色。

```
namespace Pen
{
  class Holder
  {
  public:
    Holder(HDC hdc, HPEN hPen)
      : _hdc(hdc)
    {
      _hPenOld = reinterpret_cast<HPEN>(
            ::SelectObject(_hdc, hPen));
    }
    ~Holder()
    {
      ::SelectObject(_hdc, _hPenOld);
    }
  private:
    HDC     _hdc;
    HPEN    _hPenOld;
  public:
    class White : public Win::StockObjectHolder
    ...
    class Black : public Win::StockObjectHolder
    ...
  };
}
```

注意，因为 Win::Canvas 可以隐式转换为 HDC，Pen::Color 可以隐式转换成 HPEN，所以，通过 Win::Canvas 和 Pen::Color 可以调用持有者的构造函数（一般也都是这样做的）。

使用画笔和持有者的标准方法是，在视图对象中定义并保存画笔，然后在作图过程中一直“持有”它们。下面让我们来定义两种画笔颜色：暗红和淡蓝。

```
class View
{
public:
  View()
      : _x(0), _y(0),
        _redPen(128, 0, 0),
        _bluePen(0, 0, 255)
    {}
    ...
  private:
    int _x, _y;
    Pen::Color _redPen;
    Pen::Color _bluePen;
  };
```

我们将蓝画笔进行常规绘画：

```
void View::MoveTo(Win::Canvas & canvas,
                  int x, int y, bool visible)
{
  if(visible)
  {
      Pen::Holder ph(canvas, _bluePen);
      canvas.Line(_x, _y, x, y);
  }
  PrintPos(canvas, x, y);
}
```

我们用红色画笔进行重绘：

```
void View::Paint(Win::Canvas & canvas,
                 Model::iter beg,
                 Model::iter end)
{
  PrintPos(canvas, _x, _y);
  if(beg != end)
  {
    Pen::Holder ph(canvas, _redPen);
    DrawLine draw(canvas, *beg);
    ++beg;
    std::for_each(beg, end, draw);
  }
}
```

一般来说，用户不会用肉眼区分主动画线和被动重绘线，但这里我们要尝试一下。运行一下这个程序，当改变窗口大小或者取消部分窗口覆盖时，观察一下重绘的窗口部分。你会发现，即使我们的算法是根据 WM_PAINT 消息重绘整个客户区，红线也只出现在覆盖被取消后刷新的那部分窗口。

问题在于，并不是程序绘制的所有直线都由 Windows 显示出来。例如，在客户区边界之外绘制的线永远不会显示（它们会覆盖别的窗口），Windows 把所有的图画都限制在客户区。实际上，在 WM_PAINT 消息的处理过程中，Windows 甚至会进一步限制绘图区域——扩展到无效区域中。即使在 PaintCanvas 中 Windows 只传递给我们一个方便使用的单范围矩形，而实际上无效区域可能要复杂得多（例如多重矩形），PaintCanvas 把后续所有图画都限制于这一区域之中。如果你想在这一区域之外画图，那么必须创建一个新的 UpdateCanvas。

Script 14

如果你每次画图都想用不同的颜色，那么该怎么办呢？你不可能在视图构造函数中准备好所有可能用到的画笔!你只能在要用时，创建新画笔并把它选入 DC 中，画完之后再立即取消选定它。这种方法需要一个单独的封装器 Pen::Holder::Instant 来完成。

```
namespace Pen
```

```
{
  class Holder
  {
    ...
  public:
    ...
    class Instant
    {
    public:
  Instant(HDC hdc, int r, int g, int b)
        : _hdc(hdc)
  {
    _hPen = ::CreatePen(PS_SOLID, 0, RGB(r, g, b));
    _hPenOld = reinterpret_cast<HPEN>(
                ::SelectObject(_hdc, _hPen));
  }
  ~Instant()
  {
    ::SelectObject(_hdc, _hPenOld);
    ::DeleteObject(_hPen);
  }
private:
  HDC     _hdc;
    HPEN    _hPen;
    HPEN    _hPenOld;
  };
};
}
```

我们上面所讲的画笔方法同样可应用于画刷上。画刷是用来填充区域的，例如填充实心矩形或者实心椭圆。如果把上面的程序中凡是有“pen”的地方都用“brush”代替，以上代码几乎不用再做其他任何改动就可直接应用于画刷。唯一真正的区别就是把 CreatePen API 函数换为 CreateSolidBrush 函数。除了实心画刷之外，还有阴影线画刷和图案画刷。只要相应重载 Brush::Color 构造函数就可以得到各种类型的画刷。

还有一个特殊地方要用到画刷——窗口类的定义。这里要用画刷绘制窗口的背景。例如当调用 BeginPaint 时（在 PaintCanvas 的构造函数中调用），Windows 自动重绘所有无效区域的背景。顺便说一句，EndPaint 方法使这些区域有效，所以，如果你忘记调用它，Windows 会永远不停地发送 WM_PAINT 消息。

如果要为某个类的窗口选择一种非缺省的背景颜色，你必须创建一个画刷并把它传递给 WNDCLASSEX 结构。当退出应用程序时，画刷自动被销毁（或放弃这一个类）。

```
void Win::ClassMaker::SetBkColor(int r, int g, int b)
{
  _class.hbrBackground = ::CreateSolidBrush(RGB(r, g, b));
}
```

我们在 WinMain 中调用这个方法，把窗口背景设为很浅的蓝色。

```
Win::ClassMaker winClass(className, hInst);
winClass.SetIcons(IDI_MAIN);
winClass.SetBkColor(128, 128, 255);
winClass.Register();
```

Script 15

最后，我们玩一个相似的文字游戏。我们可以改变打印文本颜色、背景颜色甚至字体。尤其是改变字体可以让我们压缩字符串。其实把字体从比例字体变为等宽字体就可以实现字符大小的缩放。应用等宽字体时，所有的字符包括空格大小都一样。我们还可以把字符串最后的无用空格删掉。

```
namespace Font
{
  class Stock: public Win::StockObject
  {
  public:
    Stock(int type): Win::StockObject(type){}
    operator HFONT()const
    {
      return reinterpret_cast<HFONT>(_obj);
    }
};

class SysFixed: public Stock
{
public:
    SysFixed(): Stock(SYSTEM_FIXED_FONT){}
};

class Holder
{
public:
  class Color
  {
  public:
    Color(HDC hdc, int r, int g, int b)
      : _hdc(hdc),
        _oldColor(::SetTextColor(_hdc, RGB(r, g, b)))
    {}
    ~Color()
    {
      ::SetTextColor(_hdc, _oldColor);
    }
  private:
```

```
  HDC           _hdc;
  COLORREF    _oldColor;
};

class Background
{
public:
  Background(HDC hdc, int r, int g, int b)
      : _hdc(hdc),
        _oldColor(::SetBkColor(_hdc, RGB(r, g, b)))
  {}
  ~Background()
  {
    ::SetBkColor(_hdc, _oldColor);
  }
private:
  HDC         _hdc;
  COLORREF    _oldColor;
};

class SysFixed : public Win::StockObjectHolder
{
public:
    SysFixed(HDC hdc)
        : Win::StockObjectHolder(hdc, SYSTEM_FIXED_FONT)
    {}
  };
};
}
```

有了上面所有这些工具，我们现在可以在深蓝色的背景下，用黄色的字、等宽系统字体显示鼠标的位置了。

```
void View::PrintPos(Win::Canvas & canvas, int x, int y)
{
  ...
  Font::Holder::Color fc(canvas, 255, 255, 0);                    // 黄
  Font::Holder::Background bk(canvas, 0, 64, 128);                // 蓝
  Font::Holder::SysFixed fix(canvas);                             // 等宽字体
  canvas.Text(0, 0, &s [0], s.length());
}
```

当然，Windows 的图形功能非常丰富，本章所讲述的只是九牛一毛，但它为读者进一步探索奠定了一个良好的基础。和前面一样，认真学习 Windows API 函数，把它那丰富的功能和数据类型应用到我们构建的面向对象框架中。

第18章 一个Windows应用程序

由于本书篇幅的限制，我无法非常深入地阐述 Windows 编程理论。下面我提纲挈领地介绍一下如何编写一个 Windows 应用程序移植到 Windows。我将把注意力主要集中在 Windows 编程的两个主要挑战上：程序的工作方式以及界面形式；怎么对不同的 API 函数进行封装和分类。Windows API 函数的数量是如此之多，对它进行某种分类（用 C++库和名字空间）是必不可少的。

我强烈建议读者在读完本章之后（以及在读的过程中），仔细阅读例子中的源代码并认真研究其中的实现细节。

18.1 把计算器移植到 Windows 环境下



Project WinCalc

把一个像符号计算器这样的命令行应用程序，通过最小的改动把它移植到 Windows 环境下是很容易实现的。我们必须找到一种除了 std::cin 之外的其他途径来获取用户的输入以及在窗口中显示结果。错误报告可以用消息框来实现。从原理上来说，一个包含两个编辑域（一个用于输入、一个用于输出）的对话框就足够了。

但是这样的移植不仅很原始，而且它不能实现原程序的所有功能。在命令行计算器中，用户至少还可以在命令窗口中看到用户以前的一些输入和计算结果（交互的历史）。另外，在 Windows 中，诸如加载、保存、退出之类的文本输入命令，现在都由菜单、按钮、工具栏或快捷键来实现了。最后，在 Windows 环境下，程序的某些功能可以得到加强，而且这也是用户的愿望。一个明显的例子就是显示计算器内存中的内容。

要实现某些上面所说的某些特点，并不需要对计算器的引擎做实质性的修改。不再使用文本输入命令实际是使解析更简单——Scanner 需要识别的标记种类变少，CommandParser

对象不再是必需的了。其他特点，比如显示内存内容，需要对 Calculator 对象做比较大的修改（它在 MVC 三元组合中扮演模型的角色）。

18.2　用户界面

不管你是进行应用程序移植，还是从头开始设计一个新程序，第一步都是规划用户界面。在本例中，显然需要一个输入域，用户在此输入要计算的表达式。还需要一个输出域，用于显示计算结果。我们还需要一个显示用户输入记录的窗口（可滚动），以及另一个显示计算器内存内容的窗口。

菜单设计不是一件无关紧要的事。菜单至少应该包括：加载（Load）、保存（Save）以及传统的关于（About）和退出（Exit）等选项。我们还可以添加另一条命令——清除内存。现在的问题是如何把这些命令组织在一起。

传统的方法遵循历史性模式，将菜单命令按照文件（File）、编辑（Edit）、视图（View）、工具（Tools）、窗口（Window）、帮助（Help）等几大项分组，这样几乎得不到便于使用的菜单系统。注意这些菜单名混合使用了名词和动词。传统方法有一定的随意性，例如，将 Exit 命令放在 File 菜单中、About 命令放在 Help 菜单中、Search 命令放在 Edit 菜单中、Option 命令放在 Tool 菜单中等。

更现代的方法是将一组命令放在一个名词标题之下，此标题描述了以上命令所作用的对象。菜单标题应该尽可能地从一般到特殊来排列。最一般的菜单项是 Program（程序），在此标题下应该放置 About 和 Exit 命令。About 命令显示程序的信息（而不是按传统方式放在 Help 菜单下）。同样，Exit 命令应该放在 Program 下，因为是退出程序而不是退出文件。

Program 后面一般是 Project、Document（或 File），然后是 View 或 Window，Edit 和 Search，以及一些特殊显示元素，如 All 或 Selection，最后是 Help。即使用户已经习惯了传统的菜单分组方式，他或她还是会发现这个新分法更有逻辑性，使用更方便。如果你想更深入学习菜单和用户界面其他元素的设计，我推荐你阅读 Alan Cooper 编撰的一本优秀著作：*About Face: The Essentials of User Interface Design*。

在计算器程序中，我让 Program 菜单包含 About 和 Exit 命令。接下来是 Memory 菜单，Memory 菜单包括 Clear、Save 以及 Load 命令（一个更完整的版本还应该包括单独的 Help 菜单）。

最后在窗口的底部加上一个状态栏，用于显示计算器是否已经准备好接受用户的输入，以及用户选择的菜单项的简要帮助。

在开始移植之前，我就不再单独列出设计草图，我将在适当时候列出最后的所有代码。如图 18-1 所示

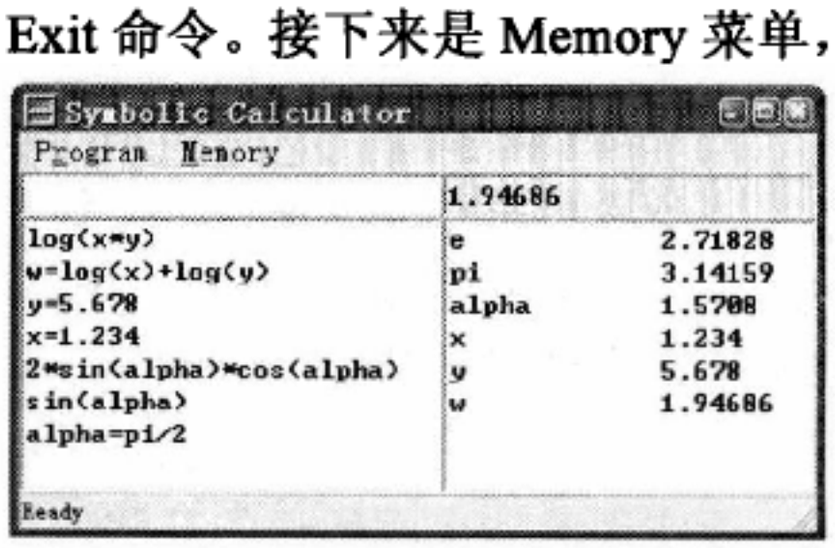


图 18-1　符号计算器的用户界面

为计算器最终用户界面。

18.3 子窗口

计算器的主窗口中有很多子窗口。其中的一些子窗口，如标题栏、菜单栏、最小化和最大化按钮等，它们都是由 Windows 创建并管理的。其他子窗口由程序员来创建和管理。图 18-2 为计算器用户界面中的一部分窗口。

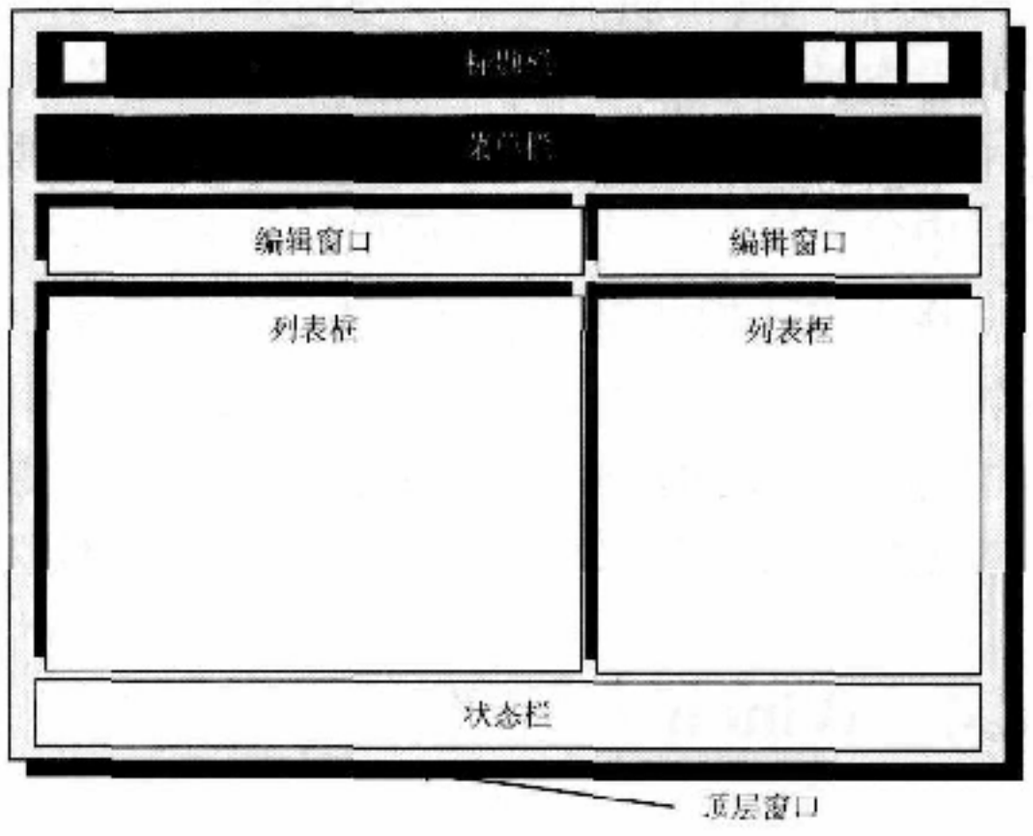


图 18-2 顶层窗口及其子窗口

子窗口的所有权在父窗口。用户在编程时经常把“子”窗口和“被拥有”窗口混为一谈，下面我们就对它们简要说明一下。

- 子窗口位于父窗口的坐标系中，并被限制在父窗口的边界之内。子窗口没有菜单，但它可以有一个整数 ID（这也是为什么 CREATESTRUCT 的菜单成员同时也作为子 ID 的原因）。要创建一个子窗口，可以把窗口风格设为 WS_CHILD，并把 CREATESTRUCT 的成员 hwndParent 初始化为它的父窗口句柄。

- 一个被拥有的窗口一般都出现在它的拥有者前面（上面），当拥有者最小化时，它也被隐藏起来，当拥有者销毁时它也被销毁。拥有者窗口的句柄由 CREATESTRUCT 的成员 hwndParent 传递。一个被拥有的窗口不一定是子窗口——在这种情况下，被拥有的窗口不使用拥有者窗口的坐标系，也不受它的边界限制。

所以如果你想让顶层窗口拥有应用程序的其他所有窗口，并且希望把它们都限制在顶层（或其他层）窗口的客户区中，你就应该把它们都创建为拥有窗口的子窗口（对话框和消息框属于被拥有但不是子窗口，我们将在稍后介绍它们）。



Script 1–2

我引入了一个单独的类 Win::ChildMaker 来封装子窗口创建操作，Win::ChildMaker 是从 Win::Maker 派生而来的。如果你知道某个窗口类的名字，你就可以创建基于它的子窗口。不过，在我们的程序中，我们将创建基于 Windows 预定义（预注册）类的窗口。

对应标准控件的窗口类中有几个很有用。通过重用这些控件，我们可以不必编写、测试新的窗口过程就得到许多标准功能。为此，我创建了一个叫做 Win::ControlMaker 的 Win::ChildMaker 子类，用于一些特殊控件生成器（比如 Win::EditMaker、Win::ListBoxMaker

和 Win::StatusBarMaker）的基类。

在详细描述控件之前，我们先了解一下子窗口的管理。我把子窗口作为视图对象的一部分，在视图对象的构造函数中调用不同的控件生成器创建不同的子窗口。在本例中有两个编辑控件，其中一个是只读的；还有两个非常适合于显示列表的列表框；最后还有一个状态栏控件。

程序要做的第一件事就是确定这些子窗口的位置。一般顶层窗口根据送过来的 WM_SIZE 消息进行子窗口的定位。在子窗口还没有显示之前，顶层窗口就会收到 WM_SIZE 消息，所以在第一次显示之前，它有充分的时间来确定子窗口的大小和位置。每次用户改变顶层窗口的大小都会产生 WM_SIZE 消息。WM_SIZE 的参数是客户区的宽度和高度。我们要做的就是在子窗口中分离这个矩形。经过简单的算术运算之后，我们把子窗口移动到新位置上。

接下来我们要讲述焦点。我们希望只要计算器被激活了，键盘输入可以直接输入到输入编辑框中。但这个不会自动发生，所以，只要计算器获得焦点，我们就把键盘焦点传递给输入编辑框。幸运的是，每次出现这种情况时，顶层窗口都会得到通知。每当子窗口被激活（包括被创建），顶层窗口都会收到 WM_SETFOCUS 消息。

18.4　Windows 控件

Windows 控件实际上是一个窗口，它的类和过程（行为的主要元素）由系统来实现。Windows 中有不同种类的控件，从非常简单的静态文本到非常复杂的 ListView 或 TreeView。它们都可以放到对话框中（我们将随后讨论），也可以创建成单独的子窗口。应用程序通过消息与控件进行通信。反过来，控件也是通过向父窗口发送消息与它通信。除了一些共同的消息之外，每个控件都有它自己独有的一套消息。

我定义了一个 Win::SimpleCtrl 类，它把所有控件相同的属性放在一起。因为控件实质上是窗口，所以 Win::SimpleCtrl 继承 Win::Dow 类而来。和所有子窗口一样，程序可以给控件分配一个数值 ID，这样父窗口就可以通过这个 ID 来区分不同的控件（通过窗口句柄也可以区分不同的控件）。如果使用适当的生成器单独创建一个控件，我们可以直接用它的句柄来初始化相应的 SimpleCtrl 对象。否则，我们就用父窗口的句柄和子窗口的 ID（用 GetDlgItem API 函数可以立即把这两者翻译成窗口句柄）来进行初始化。

18.4.1　静态文本

最简单的控件叫做静态文本，它用一个简单框架显示文本。它对应的对象 Win::StaticText 是 Win::SimpleCtrl 最简单的后代。通过调用 Win::StaticText 从 Win::Dow 继承过来的 SetText 方法，程序可以改变显示的文本内容。

18.4.2　编辑控件

编辑控件稍微复杂一些。只读编辑控件除了可以把它的内容拷贝到剪贴板之外（选中要

拷贝的文本，然后按组合键 Ctrl+C），其他和静态文本一样。完整功能的编辑控件不仅能显示文本，还可以读取用户的输入。它还支持几种编辑功能，如删除、拷贝、剪切、粘贴等。实际上，一个多行编辑控件使用就是 Windows 记事本的引擎。

对一个程序来说，最想从编辑控件得到的是用户输入的文本。Win::SimpleCtrl 的另外一个后代类 Win::Edit 提供了两个方法来获取用户的输入。底层 GetText 方法的参数包括一个缓存器和一个计数（你可以先调用 GetLen 方法查询文本的长度）。顶层 GetText 方法没有参数，它返回一个 std::string。

控件不断地向父窗口发送消息。其中比较有趣的是 WM_COMMAND，针对新一代控件就是 WM_NOTIFY。为什么会这样呢？由于历史原因 WM_COMMAND 并不是控件专用的，菜单和加速键都使用 WM_COMMAND 消息。通用窗口过程将它们区分开来，把它们分别叫作 OnControl 和 OnCommand。OnControl 传递控件的窗口句柄、数值 ID 以及命令的 ID。例如，每当用户改变编辑控件中的文本内容时，它都会向父窗口发送一个命令 ID 为 EN_CHANGE 的命令消息（应用程序通常忽略此消息）。

为了使控件正确工作，我们必须在主消息循环中增加对 TranslateMessage 函数的调用。

```
MSG  msg;
int status;
while((status = ::GetMessage(& msg, 0, 0, 0))!= 0)
{
  if(status == -1)
    return -1;
  ::TranslateMessage(& msg);
  ::DispatchMessage(& msg);
}
return msg.wParam;
```

此 API 函数把原始键盘输入翻译成命令和字符。如果没有它，我们的编辑控件就无法显示从键盘输入的字符。

18.4.3 窗口子类

编辑控件的主要问题是：什么时候是读取其中文本的最佳时机？用户必须通过某种方式告诉程序输入是否完成，在我们的程序中，就是整个表达式都输入完毕时。我们可以添加一个特殊的“求值”按钮，当用户点击它的时候就计算所输入表达式的值。但这不是用户所希望的，他们希望只要按一下键盘上的回车键，程序就知道表达式输入完毕。但令人奇怪的是，看起来这么简单的一个功能实现起来并不容易。没有一个简单的方法供编辑控件来通知父窗口回车键按下了。

我们该怎么办呢？看样子只好用窗口子类这个后门了。编写自己的窗口过程把它插入到编辑控件中。不过幸运的是我们的窗口过程不必从头实现所有的编辑功能，它要做的只是拦

截几个我们感兴趣的消息，而把其他消息还是传递给原来的 Windows 定义过程（如图 18-3 所示）。

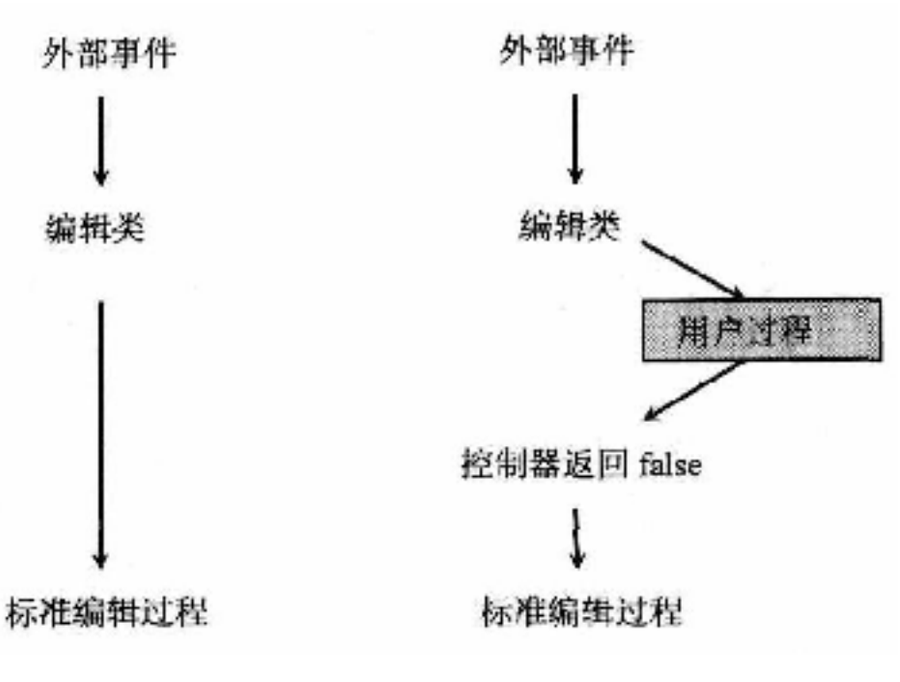


图 18-3　通过插入一个用户自定义过程取 Windows 编辑控件的子类

Script 3–4

窗口子类很方便地封装在两个 Win::Dow 方法中：SubClass 和 UnSubClass。SubClass 方法带有一个 SubController 对象的指针。它取代了旧窗口过程中一般性 SubProcedure 方法的位置。SubProcedure 首先调用 SubController 处理所有的消息。如果 SubController 没有处理消息（也就是说它的返回值为 false），那么它就调用旧的窗口过程。注意这些过程是如何链接在一起的。每一个过程或者进行消息处理或者调用下一个过程，直到最后一个过程调用缺省的窗口过程。

对编辑控件取子类的过程是这样的。首先我们定义一个 Win::SubController 的子类 EditController。它只覆盖一个方法 OnKeyDown，仅当回车键（VK_RETURN）按下时才处理键盘输入。拦截了回车键之后，EditController 向父窗口发送一条消息。此处，我决定使用一条标准控件消息 WM_CONTROL，父窗口的 OnControl 方法可以处理这条消息。我必须打包 wParam 和 lParam 参数来模仿 Windows 发送的控件消息。我选用预定义常量 IDOK 作为控件 ID（模拟按下对话框中的 OK 按钮——参见本章后面的“对话框”部分内容）。

我们调用编辑窗口的 SubClass 方法，并把它传递给 EditController，真正执行子类操作。

你也许会对 Windows 中使用明显的面向对象术语感到好奇。虽然 Windows API 函数不是面向对象的，但它还是使用如窗口类、子类等等这样一些术语。你必须清楚，图形化操作系统并不是由 Microsoft 或者 Apple 发明的，它最初是为 Xerox PARC 设计的。第一个实现窗口系统的是 Smalltalk——它是面向对象语言的典范。所以窗口类实际上是以 Smalltalk 类的方式实现的。它有自己的数据成员和方法。在 Windows 中，窗口过程就好像一个“方法”一样。Get/SetWindowLong API 用于访问“数据成员”。子类意味着覆盖虚函数——在 Windows 中，就是覆盖窗口过程。

18.4.4　插入计算器

Script 5

现在我们必须覆盖主控制器的 OnControl 方法，以截取 IDOK 消息并作出相应的反应。

事实上，这也是原来 main 过程的所有代码要实现的目标。我们从编辑控件中读取用户输入的字符串，把它封装到一个流中，创建扫描器和解析器，然后计算其结果。此时，显示结果并更新历史记录的显示。

嵌入对象计算器对象的最佳位置是在 TopController 对象中。它相当于模型—视图—控制器三元组合中的模型部分。

现在是时候做一些 Calculator 清理工作了。首先把 cerr 和<iostream>清除掉（<istream>要留着，在扫描器的实现里还要使用它）。计算器中的某些错误转化为异常——我们会在 Syntax 异常中嵌入字符使之描述更清楚。对于其他不会影响计算准确性的错误，我们把它们重定向到全局对象 TheOutput，它会用消息框显示这些错误。

另外，现在有了菜单系统，我们可以把命令解析语句去掉了。

18.4.5 列表框控件

显示历史记录非常简单。当用户输入完成后，我们只要告诉列表框在其窗口添加一行就行了。列表框会自动存储和显示字符串，必要时还能滚动和重绘。这些事情都由 Windows 控件完成，我们无需做任何事。

而要显示内存内容并不很容易。因为在这种情况下，显示的结果必须反映不断变化的数据结构状态。在用户操作过程中，一方面要增添新变量，另外一方面要修改旧变量的值。只有解析器才能理解这些操作。由它判断用户在输入窗口中输入的的表达式是否改变了计算器内存的内容。于是，显示也必须随着模型内部的改变而改变。视图需要模型的反馈信息。

18.4.6 模块—视图反馈环

有两种处理模型—视图反馈环的方式。第一种方式比较简单，它假设每个可能改变模型状态的用户动作都需要刷新对应视图部分。根据这一方案，显示的改变是受控制器驱动的。控制器通知视图进行刷新，然后视图向模型查询要改变的显示。模型与视图没有依附关系。

第二种通知方式假设模型直接通知视图。这个通知可能仅仅通知视图进行刷新，也可能对刷新内容和刷新方式提供准确信息。这种方式的问题在于，它会带来视图和模型循环依附的问题。视图依赖于模型，因为它必须知道如何查询即将显示的数据。模型也依赖于视图，因为它必须知道如何通知所发生的变化。如果你回过头看看模型的定义，你会发现，模型并不知道用户界面的细节。如果我们使用这种方法，我们就不得不放弃原来明确的责任划分，打破简单的依附关系层次，转向一个依附关系错综复杂的系统。

不要绝望！我会告诉你如何既能使用通知又能保持简单的层次关系。这次使用的技巧是只允许模型看见视图的很小一部分。模型所需要的是一个通知接收器，一个对某些特定事件的通知感兴趣的对象。模型不需要知道这些通知如何使用，并且它也不知道类 View 是

否存在。

Script 6

通知接收器是一个只封装了通知接口的单独类。模型必须知道这个类的接口，可以对这个类的对象进行访问，还可以调用它的方法。视图继承通知接收器类并实现它自己的虚方法。当模型向它的通知接收器对象发送通知时，实际上是把通知发送给了视图。但它仍然不知道用户界面（UI）的细节（如图 18-4 所示）。

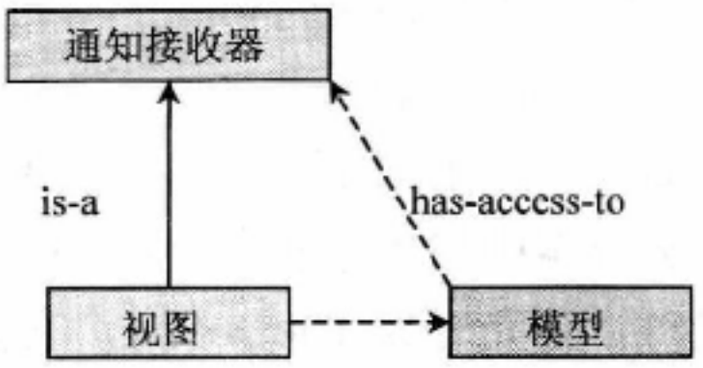


图 18-4　引入通知接收器分解依赖性循环

我们可以通过包含层次知道在这种方法中没有依附循环。定义通知接收器的头文件位于最顶层。它必须包含于 View.h，因为 View 类继承于类 NotificationSink。构成模型的许多文件（Calculator 对象）也必须包含它，因为它们要调用 NotificationSink 的方法。View.cpp 必须包含 Calc.h，因为它需要来自模型的数据。还有重要的一点是，模型文件不能包含 View.h。这种依赖关系图是一个 DAG（directed acyclic graph，有向非循环图），DAG 是良好设计的标志。

NotificationSink 接口是一个抽象类：

```
class NotificationSink
{
public:
  virtual void AddItem(int id)= 0;
  virtual void UpdateItem(int id)= 0;
};
```

NotificationSink 使用数值 ID 与内存项（变量）对话。通知的接收者（接收器的实际实现者）必须通过这些 ID 读取内存数据（变量的名字和当前值）。

Script 7

下面是 View 实现这两个方法的代码。AddItem 给列表框添加一个新行（由 FormatMemoryString 格式化）。

```
void View::AddItem(int id)
{
  std::string varStr = FormatMemoryString(id);
  int i = _memoryView.AddString(varStr);
  _memoryView.SetData(i, id);
}
```

注意，在程序中我们使用了列表框能够在每行添加隐藏数据项的特点。我们在每一行都添加了变量的 ID，这样就可以显示变量的名字和值。这使得 UpdateItem 的实现变得简单

多了。

```
void View::UpdateItem(int id)
{
  int count = _memoryView.GetCount();
  int i;
  for(i = 0; i < count; ++i)
  {
    int idDisp = _memoryView.GetData(i);
    if(id == idDisp)
    {
      std::string varStr = FormatMemoryString(id);
      _memoryView.ReplaceString(i, varStr);
      _memoryView.SetData(i, id);
      break;
    }
  }
  assert(i != count); // 找到了!
}
```

内存列表框每行都必须显示变量的名字和值，我们把列表框的每一行都分成两列显示。

```
std::string View::FormatMemoryString(int id)
{
  std::string varStr = _calc.GetVariableName(id);
  varStr.resize(_varNameChars, ' ');
  varStr += ' ';
  double x;
  if(_calc.GetVariableValue(id, x))
    varStr.append(ToString(x));
  else
    varStr.append("?");
  return varStr;
}
```

注意这一实现方式所需的两个新 Calculator 方法。

这种格式化带来的副作用是，每当列表框的大小改变时，我们必须重新显示一遍——因为两列的宽度都改变了。通过增加对 Size 方法的两次调用来实现这一点。

```
void View::Size(int width, int height)
{
  _memoryView.Clear();
  ...do the resizing work
  _calc.RefreshMemory();
}
```

实际上所有的复杂操作都由 Calculator 对象完成。它的 RefreshMemory 方法告诉 Store 检查所有的变量，并把 AddItem 通知发送给 NotificationSink。这技巧不仅简化了内存的显示刷新，还在应用程序启动时完成了它的初始化。WM_SIZE 是 WM_CREATE 之后第一批发送给顶层窗口的消息之一。OnCreate 方法确保一切都准备完毕。

```
bool TopController::OnCreate(
  Win::CreateData const * create, bool & success)throw()
{
try
{
  _view.reset(new View(_h, _calc));
  _calc.SetNotificationSink(_view.get());
  success = true;
}
catch(...)
{
  TheOutput.Error(
      "Internal Error:\nFailure to initialize.");
}
return true;
}
```

最后，我们举一个计算器使用消息接收器的例子：

```
int SymbolTable::ForceAdd(std::string const & str)
{
  _dictionary [str] = _id;
  // Notify of new item
  if(_sink)
    _sink->AddItem(_id);
  return _id++;
}
```

18.5　命令和菜单

成员指针。

Script 8

在命令行程序版本中，我们必须使用一个特殊的解析器来解码用户命令。在 Windows 中就简单多了。命令主要是通过选择菜单输入的。这样做有个明显的好处，用户不必把每条命令的名字和语法都记得清清楚楚。在菜单系统中，所有命令都经过分门别类精心组织。即使

程序可以通过其他途径输入命令，比如快捷键和工具栏，菜单在教给初学者程序的功能方面仍然扮演重要的角色。

在 Windows 程序中实现菜单相对来说比较简单。你可以在资源脚本中定义菜单——通过资源编辑器可以轻松完成。例如，计算器的菜单在 script.rc 中的描述如下所示：

```
ID_MENU MENU DISCARDABLE
BEGIN
POPUP "&Program"
BEGIN
MENUITEM "&About...",     ID_PROGRAM_ABOUT
MENUITEM SEPARATOR
MENUITEM "E&xit",         ID_PROGRAM_EXIT
END
POPUP "&Memory"
BEGIN
MENUITEM "&Clear",            ID_MEMORY_CLEAR
MENUITEM SEPARATOR
MENUITEM "&Save...",          ID_MEMORY_SAVE
MENUITEM "&Load...",          ID_MEMORY_LOAD
END
END
```

请注意，每一个菜单项的名字中都嵌入了一个“&”符号。Windows 将&翻译成键盘的快捷键（不要与加速键混淆）。“&”本身并不显示，它后面的字符以下划线格式显示。用户在按下 Alt 键后，同时按下这个键就可以直接选中对应的菜单项。

你可以给菜单取个名字或者给它分配一个数值 ID（本例中为 ID_MENU），然后，把它传递给 Win::ClassMaker 或每个单独的 Win::Maker。

每个菜单项都有一个唯一的 ID，一般把它叫做命令 ID（例如 ID_PROGRAM_ABOUT）。此 ID 是 WM_COMMAND 消息的参数之一，用户不论何时选择了某个菜单项，Windows 都会发出 WM_COMMAND 消息。你所要做的就是实现顶层控制器的 OnCommand 方法来对这些命令作出响应。

实现 OnCommand 最简单的方法就是编写一段庞大的 switch 语句，每条命令对应一个 case 语句，不过这样就显得很繁琐。难道就没有其他更好的方法了吗？我们能不能像窗口过程那样，把 switch 语句隐藏在库中呢？不能这样做——与 Windows 消息不一样，因为 Windows 消息是由 Windows 预先定义，以后不会改变，而不同的用户自定义菜单提供的命令和命令 ID 完全不同。

另一方面，菜单命令不会提供变量数量和参数类型。事实上，菜单命令没有任何参数——如果需要参数，以后由对话框输入。那么，我们为什么不按计算器中组织内置函数的方式，把所有的命令放到一个不带任何参数的函数指针数组中呢？

这种方式只有一个问题——为了使用函数指针，命令必须实现为自由函数。但是我们知道，大部分命令都要访问模型（Calculator）。我们不想把 Calculator 设为全局对象，让命令知道它的名字。对全局对象的隐式依赖是不良设计的标志，如果在复杂程序中出现这样的情况，那么对它的维护将异常艰难。

那么，创建一个单独的对象 Commander，把我们的命令作为它的方法行不行呢？这个对象可以是 TopController 的成员，并在控制器的 OnCreate 方法中初始化。它不必被设为全局对象就可以轻易访问 Calculator。我们可以创建一组 Commander 方法的指针矢量，并使用它分配菜单命令（参见图 18-5）。这和我们在计算器中分配函数调用的方案一样。

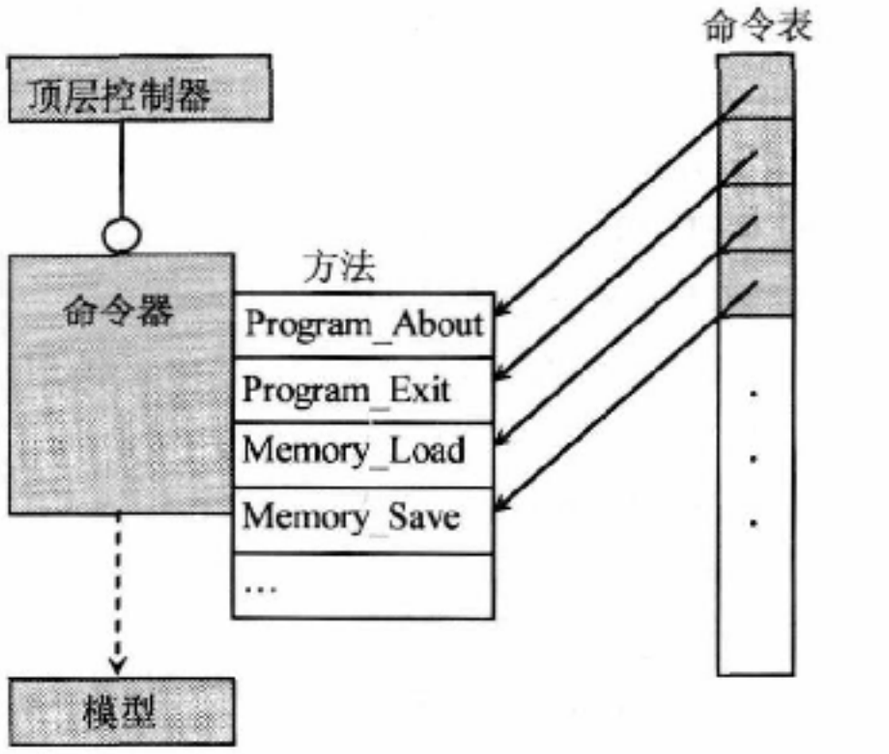


图 18-5　Commander 对象及一个指向其方法的指针的矢量

但是指向方法的指针是什么呢？和自由函数不一样的是，一个（非静态）方法只能在对象中调用，这一对象成为 this 指针的提供者。另外，方法指针的定义还必须指定某个类，指针可能指向这一个类的方法。一个 Controller 方法的指针与一个 Commander 方法的指针完全不一样。在成员指针的声明中反映了这一要求。例如，一个指向 Commander 方法 pCmdMethod 的指针没有参数，也没有返回值，如下所示：

```
void(Commander::*pCmdMethod)();
```

可以把这样一个指针初始化为指向 Commander 对象的某个特定方法，例如 Program_About。

```
pCmdMethod = &Commander::Program_About;
```

假定 Commander 类有一个对象_commander，我们可以通过成员指针调用它的方法。

```
(_commander.*pCmdMethod)();
```

如果_commander 是一个指针而不是一个对象（或引用），上面的语句应该改成：

```
(_commander->*pCmdMethod)();
```

我们将把所有的命令定义为 Commander 的成员，并赋予它们一个与菜单中位置对应的名字。命令矢量的定义和初始化如下所示：

```
*typedef void(Commander::*CmdMethod)();
const CmdMethod CmdTable [] =
{
    &Commander::Program_About,
    &Commander::Program_Exit,
    &Commander::Memory_Clear,
    &Commander::Memory_Save,
    &Commander::Memory_Load,
```

```
        0 // Sentinel
    };
```

那么我们的 CmdTable 在菜单系统下有效吗？当用户选择某个菜单项时会发生什么呢？首先，一般性窗口过程会收到 WM_COMMAND 消息，它会调用 TopController 的 OnCommand 方法。这个方法会把命令 ID（在资源脚本文件中与菜单一起定义）翻译成 CmdTable 中相应的索引，然后执行对应的方法。

```
(_commander->*CmdTable [idx])();
```

把命令 ID 翻译成索引是这个方法中最薄弱的环节。实际上，在资源文件中定义菜单并不像想象的那么简单。

Script 9

一个中等复杂的应用程序可能会根据当前程序状态改变菜单。最简单的例子就是计算器菜单中的 Memory>Save 菜单项。当用户还没有给内存添加一个新变量时，它应该是无效的（灰色显示）。而当用户增加了新变量后，应该重新激活这一菜单项。但这就要求模型对用户界面（菜单）有所了解。我们仍然可以利用通知接收器保存这一状态。不过，有一种更好更一般的途径——使用动态菜单。

18.6 动态菜单

首先，我们对命令表的思想作一个总结并加以扩展。根据前面的内容可知，要实现完整的菜单就必需一个成员函数指针数组。对每个命令项，我们可以自由添加新的数据（和功能）。除了可以执行给定命令的成员指针外，让我们再添加一个可以快速测试命令可用性的成员指针。例如，可用性信息可用于动态禁止（变灰）某些菜单项或其他控件。为每条命令添加对应的简短帮助字符串，也可以方便用户的使用。我们在状态栏显示此帮助字符串。

我还决定用字符串名与命令相关联，而不是用整数标识符，这种做法更为普遍。尽管通过 ID 搜索命令项的速度比通过字符串搜索要快，但如果命令项不是太多，这种速度差异不会明显。而且，当程序发展到不仅包含菜单还添加了加速键和工具栏后，通过名字来指定命令比用 ID 更容易维护。

Script 10

下面是一个（可再利用）命令项的新定义——这是命令表的基础。

```
namespace Cmd
{
  template <class T>
  class Item
```

```
    {
    public:
      char const * _name;              // 正式名
      void(T::*_exec)();               // 执行命令
      Status(T::*_test)()const;        // 测试命令状态
      char const * _help;              // 帮助字符串
    };
}
```

如果我们想重用 Cmd::Item，就必须把它做成一个模板。模板的参数是特定 commander 类，其中包含我们需要访问的方法。

下面是 Windows 库的客户创建静态命令表并用适当字符串及成员指针进行初始化的方式。

```
namespace Cmd
{
  const Cmd::Item<Commander> Table [] =
  {
    { "Program_About",    &Commander::Program_About,
                          &Commander::can_Program_About,
                          "About this program"},
    { "Program_Exit",     &Commander::Program_Exit,
                          &Commander::can_Program_Exit,
                          "Exit program"},
    { "Memory_Clear",     &Commander::Memory_Clear,
                          &Commander::can_Memory_Clear,
                          "Clear memory"},
    { "Memory_Save",      &Commander::Memory_Save,
                          &Commander::can_Memory_Save,
                          "Save memory to file"},
    { "Memory_Load",      &Commander::Memory_Load,
                          &Commander::can_Memory_Load,
                          "Load memory from file"},
    { 0, 0, 0, 0}
    };
}
```

这里 Commander 是计算器使用的命令器 Commander 类的名字。

命令表用来初始化实际的命令矢量 Cmd::VectorExec，它可以增加此数据结构的功能。Cmd::Table 和 Cmd::VectorExec 之间的关系与计算器中的 Function::Array 和 Function::Table 之间的关系相似。和前面一样，使用这种方法可以很容易地在表中添加新项目——即向程序中添加新菜单命令。

```
namespace Cmd
{
  template <class T>
  class VectorExec: public Vector
  {
  public:
    VectorExec(Cmd::Item<T> const * cmdTable,
               T * commander);
    void Execute(int cmdId)const;
    Status Test(char const * cmdName)const;
    char const * GetHelp(int cmdId)const;
  protected:
    T *                    _commander;
    Cmd::Item<T> const *  _cmd;
  };
}
```

与 Cmd::Item 一样，Cmd::VectorExec 必须是模板。为了不至于模板化使用此矢量的所有对象（尤其是菜单系统），我让它派生于一个非模板类 Cmd::Vector。这个类定义了一些纯虚函数和一些一般性功能，如通过使用映射的名字来搜索命令。

```
namespace Cmd
{
  class Vector
  {
  public:
    virtual ~Vector(){}
    int CmdToId(char const * cmdName)const;
    virtual void Execute(int cmdId)const = 0;
    virtual Status Test(char const * cmdName)const = 0;
  protected:
    typedef map<char const *, int, NocaseCmp> CmdMap;
    CmdMap _cmdMap; // 名字到 ID
  };
}
```

图 18-6 所示为命令矢量派生和初始化中不同类和对象间的关系。让我们重温这种关系。Cmd::Vector 是派生 Cmd::VectorExec 模板的基类。库的客户使用他们自己的 Commander 类实例化这个模板。一旦某个 Cmd::VectorExec 类型的对象初始化完成，客户就把它作为一个 Cmd::Vector。我们稍后就会看到，库的其他组件也是根据 Cmd::Vector 接口编写（这样，它们就不必对于另外一些 Controller 类进行模板化）。客户的 Cmd::VectorExec 对象初始化使用一个静态 Cmd::Table 来完成，这个静态 Cmd::Table 是一个 Cmd::Items 数组。

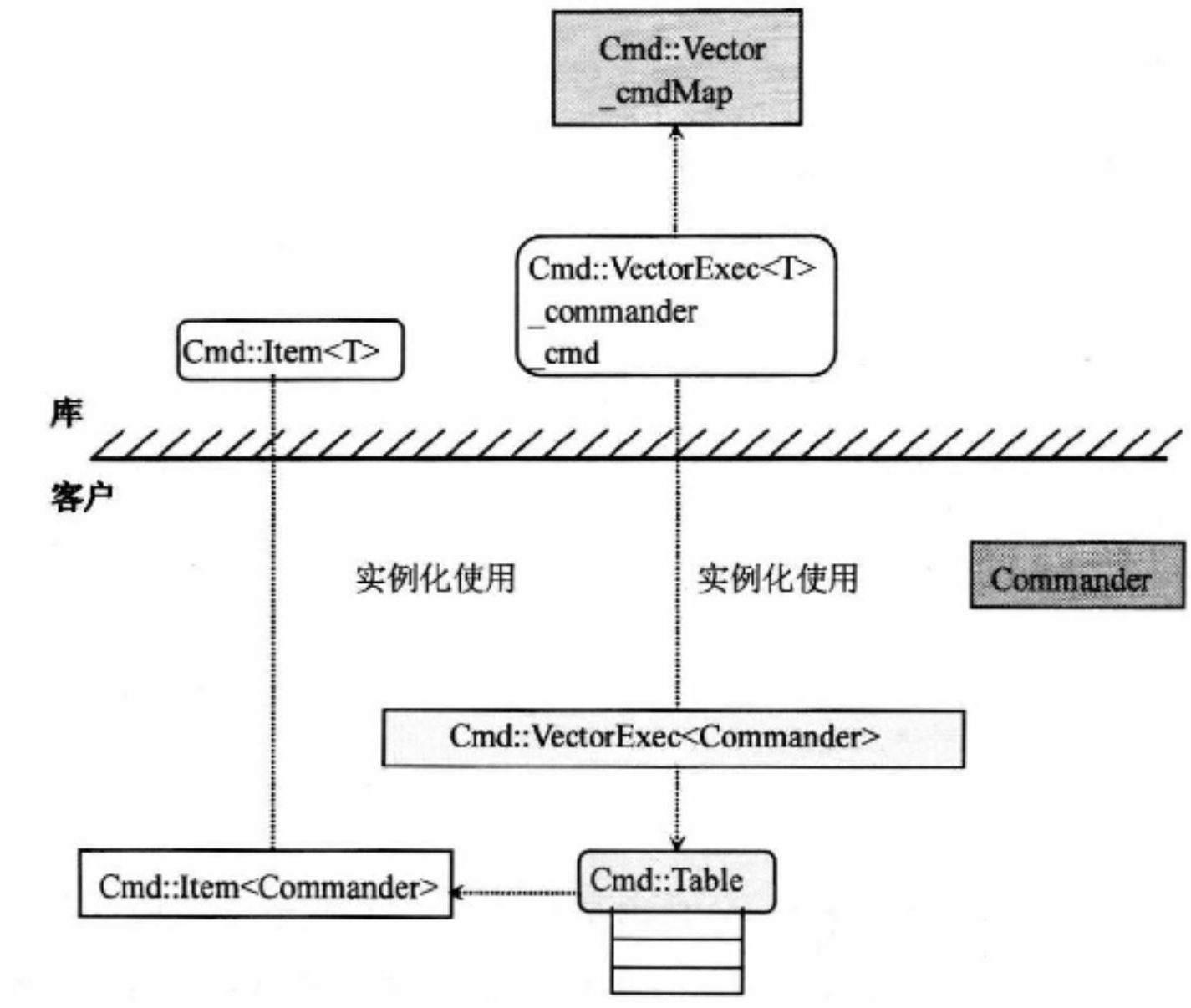


图 18-6　在实现命令矢量中库类与客户类的关系

Script 11

菜单提供对命令矢量的访问。在一个动态菜单系统中，我们用一个静态表来初始化菜单。这个表是按层次组织的：菜单项指向包含命令的弹出菜单。例如，下面是我们的计算器菜单的初始化表。

```
namespace Menu
{
  const Item programItems [] =
  {
    {CMD,       "&About...", "Program_About"},
    {SEPARATOR,  0,          0},
    {CMD,       "E&xit",     "Program_Exit"},
    {END,       0,           0}
  };
  const Item memoryItems [] =
  {
    {CMD,       "&Clear",    "Memory_Clear"},
    {SEPARATOR,  0,          0},
    {CMD,       "&Save...",  "Memory_Save"},
    {CMD,       "&Load...",  "Memory_Load"},
    {END,       0,           0}
```

```
};
//--- /---- 菜单栏 ---- / ----
const BarItem barItems [] =
{
  {POP,   "P&rogram", "Program",   programItems},
  {POP,   "&Memory",  "Memory",    memoryItems},
  {END,   0,          0,           0}
};
}
```

注意，对于名字后带省略号（...）的命令，还需要用户通过对话框等输入进一步的信息。所有的菜单项同时指定了对应的命令名字——对于弹出项来说，这些字符串与命令名字相同。菜单栏上的菜单项也有名字，不过没有命令与它们关联。菜单栏上的菜单项具有对应于弹出菜单表的指针。顺便说一句，加速键和工具栏也可以用类似的表进行初始化。

注意 Menu::Item 类和先前介绍的 Cmd::Item 没有任何关系。我们将名字空间修饰符作为类名的一部分。如果不使用名字空间，那这两个类的名字可能会是 MenuItem 和 CmdItem。

Menu::DropDown 类的实际菜单对象是在 View 的构造函数中创建。它由上面所示的菜单栏项目表 Menu::barItems 和一个 Cmd::Vector 对象（用 Cmd::Table 对它进行初始化）初始化（参见图 18-7）。其他部分可以很方便地封装到 Windows 库中。

你可能感兴趣的是：既然菜单是一种资源（使用 DestroyMenu API 释放），那么 Menu::Maker 类应该还有转移功能。例如，当我们创建一个菜单栏时，所有的弹出菜单一个接一个地被转移到 Menu::BarMaker。

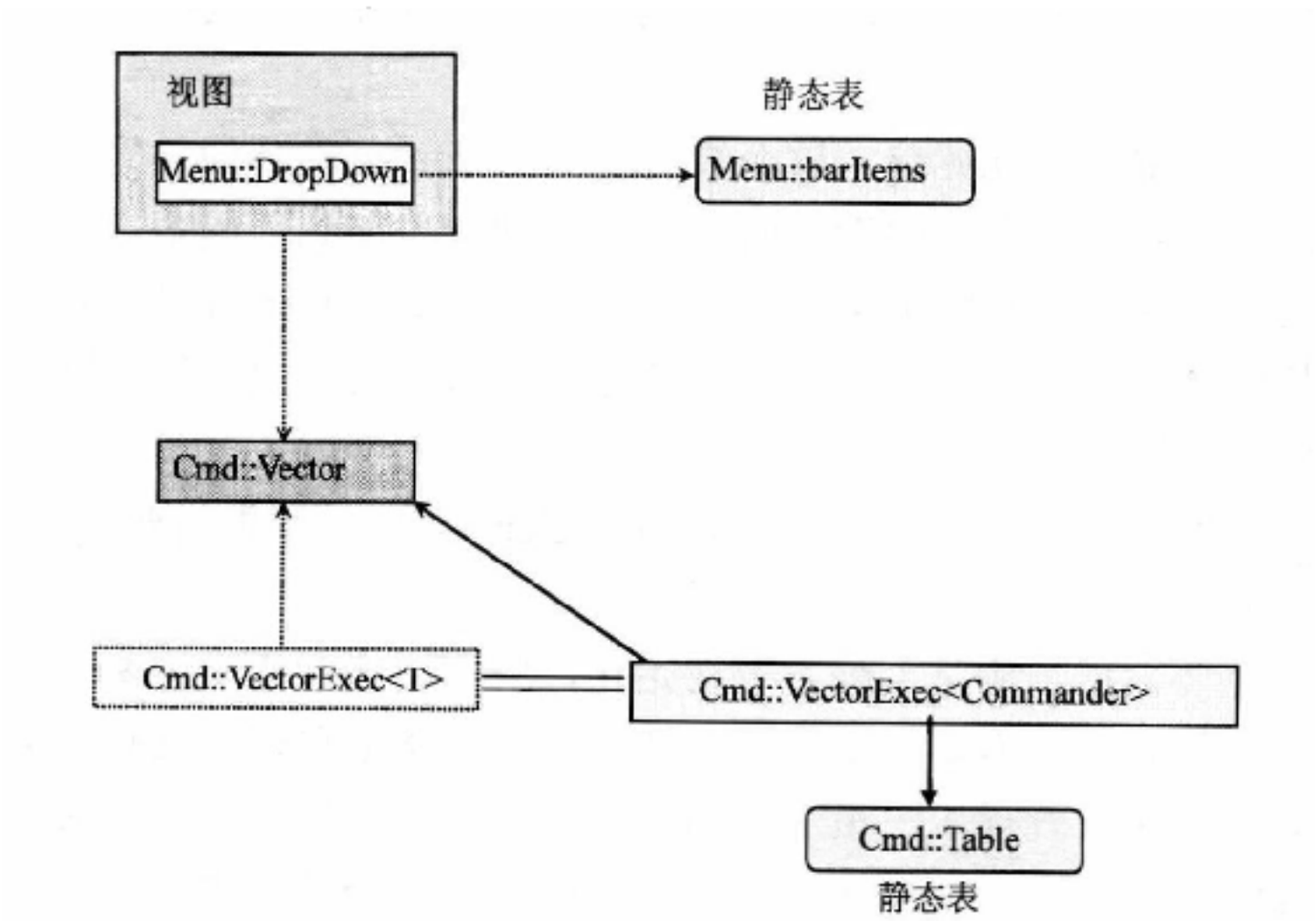


图 18-7　下拉菜单和相关类的初始化



Script 12

至此，我们的工作还没有结束。我们希望能动态激活和禁止特定的菜单项。现在我们已经有了测试某个命令是否有效的 Commander 方法——实际上，这些方法要通过命令矢量访问。现在的问题是，什么时候是调用这些方法的最佳时机？Windows 在打开一个弹出菜单前，它会发出一条消息 WM_INITMENUPOPUP。这一消息的处理器叫做 OnInitPopup。我们可以利用测试菜单可用性的这一时机来操纵菜单。事实上，既然库类 Menu::DropDown 可以访问命令矢量，也就可以实现 RefreshPopup 方法。库的客户不必再编写其他任何代码。

显示选中菜单项的简短帮助也很容易。当用户把鼠标移到弹出菜单项上时，Windows 会给我们发送一条消息 WM_MENUSELECT，我们可以在控制器方法 OnMenuSelect 中对这条消息进行处理。我们所要做的就是调用命令矢量的 GetHelp 方法，并在状态栏上显示菜单的帮助信息。

下面我们从 Windows 库客户的角度来重新审视一下整个任务的未完成过程。如果客户要使用我们的动态菜单系统，他需要编写一些什么代码呢？首先，他必须实现命令器（Commander），命令器是程序中所有有效命令的仓库。对每一条命令必须实现两个方法：一个用来执行命令本身，另一个用来测试它的可用性。

命令器的作用如下所示：

- 获取用户输入的数据（如果需要的话），这个操作通常由对话框完成。
- 把请求分配给模型执行。

完成命令器之后，客户必须创建一个静态命令表，并对它进行静态初始化。给这个表中的所有命令都分配一个名字和简短的帮助字符串。然后使用这个表对命令矢量进行初始化。

菜单系统同样也由静态表初始化。这个表包括命令名、菜单项的显示名以及区分命令、分隔符和栏项目的标记。一旦菜单栏准备就绪，就要把它添加到顶层窗口上。不过，不要在视图的构造函数内添加菜单。在添加菜单之前，View 和 Controller 都必须完全构建完毕。因为菜单的添加会导致向顶层窗口发送一系列的消息(特别是,要求改变客户区域大小的消息)，所以控制器必须准备好按一种有序方式来处理这些消息。

最后，用户还要实现 OnInitPopup 和 OnMenuSelect 方法（如果需要的话），分别用于刷新弹出菜单和显示简短帮助信息。

因为菜单中的主要数据结构是由静态表初始化，所以更改起来很容易。例如，重新排列菜单或者改变菜单项的名字只需要改变一个文件——即包含菜单表的文件。要改变命令的行为，只需要改变命令器对象就行了。最后，增加一个新命令可以分为三个独立的阶段：给命令器添加适当的方法；在命令表中添加一个入口；给菜单表添加一个命令项。以上步骤非常简单且不容易出错。

图 18-8 显示了控制器中不同元素之间的关系的依赖性。

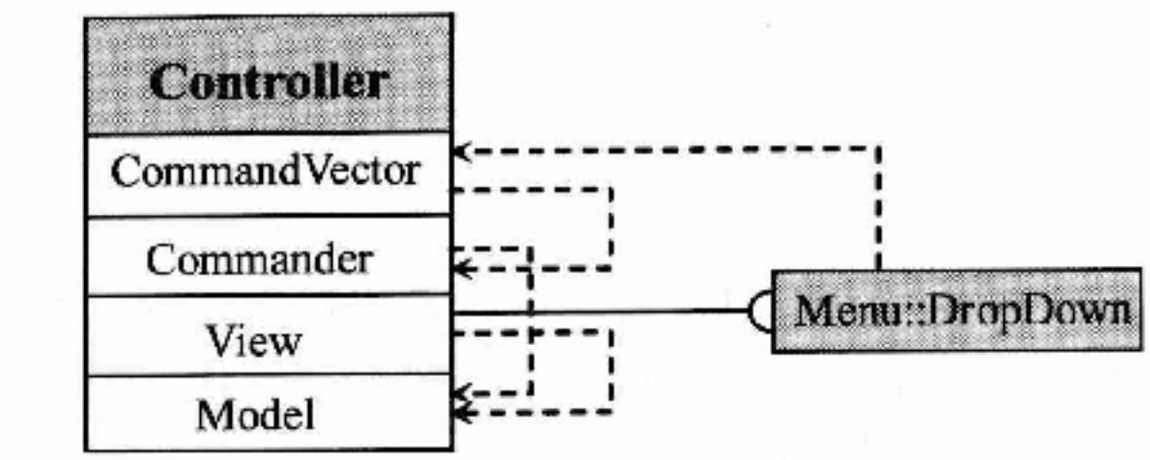


图 18-8 控制器不同元素间的关系

Script 13

因为 Commander 不能(也不应该)直接访问 View,所以在 Memory_Clear 和 Memory_Load 命令执行完之后，没有什么简单的方法可以强制刷新显示。这样我们只好使用强制手段（在每条命令执行完之后都刷新内存显示）或者使用通知。我决定使用更一般的通知机制——发送 Windows 消息。为了强制清除计算器的内存显示，Commander 要向顶层窗口发送一条用户自定义消息 UM_MEMCLEAR。

记住，消息其实就是一个数。你可以自由定义自己的消息，只要你分配给它的消息号不和 Windows 定义的消息号冲突就行。使用特殊标识符 WM_USER 定义消息号，我们可以确保此消息号大于任何 Windows 定义的消息号。

我给 Win::Controller 新增了一个消息处理器 OnUserMessage 来处理用户自定义消息。只要当消息号大于或等于 WM_USER 时，就调用这个处理器来处理消息。

要使菜单能正确工作还有一件事要做。在消息循环中，必须在 DispatchMessage 之前调用 TranslateMessage。其实在前面，为了使嵌入控件能正确工作，我们已经修改了消息循环。这次改动的基础是：TranslateMessage 可以过滤那些必须翻译成菜单快捷键的键盘消息，并把它们转换为 WM_COMMAND 消息。

如果你还想加入键盘加速键（不要和键盘快捷键混淆起来，快捷键是由菜单系统直接处理）。例如，Ctrl+L 表示装载内存——你必须进一步扩充消息循环，加入 TranslateAccelerator 的调用。

Script 14

虽然我们不想在这里讨论无模式对话框，但是你可能会有兴趣知道，非模态对话框也需要一个预处理步骤，在消息循环中调用 IsDialogMessage。把所有的这些加速键和无模式对话框句柄添加到类 Win::MessagePrepro 的一个单独预处理器对象上是很有意义的。它使用 Pump 方法进入消息循环，仅当顶层窗口销毁时才返回。把这些预处理器对象传递给顶层控制器，使得动态转换加速键表或者创建和销毁无模式对话框成为可能。

18.7　对话框

Script 15

实际上，Windows 控件更普遍的应用是用来构建对话框。对话框是一个预建的窗口，它为程序员指定的不同控件提供了框架。控件的类型和位置一般都是在资源脚本中定义（和图标一样）。下面就是这样描述脚本的一个例子：

```
#include <winres.h>
IDD_ABOUT DIALOG DISCARDABLE  0, 0, 142, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "About Symbolic Calculator"
FONT 8, "MS Sans Serif"
BEGIN
  DEFPUSHBUTTON   "OK",IDOK,46,49,50,14
  CTEXT           "Bartosz Milewski © 2000",
                    IDC_STATIC,10,33,121,14
  ICON            IDI_MAIN,IDC_STATIC,60,7,20,20
END
```

你不必学习资源脚本语言，因为一般资源编辑器会自动生成资源脚本。上面的脚本文件所对应的对话框如图 18-9 所示。

这个对话框包含三个控件：一个缺省形式的按钮、一个静态文本和一个静态图标。按钮的 ID 是 IDOK，其他两个静态控件的 ID 相同，都是 IDC_STATIC。这些特殊的 ID 都是预定义的。不过，通常只要你在资源头文件（Resource.h）中定义了它们的符号，你就可以自由地给控件定义自己的 ID。

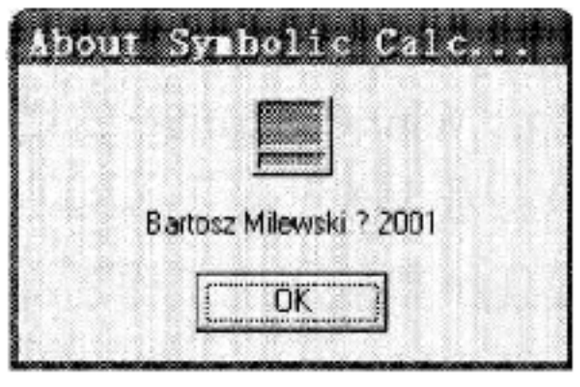


图 18-9　符号计算器“About”对话框

Load 和 Save 对话框（在 Memory 菜单项中选择 Load 和 Save 后会出现这两个对话框——我们很快就会讲到）中的东西稍微多一些。它们都有一个 OK 按钮和一个 Cancel 按钮。Save 对话框还包含一个编辑控件，Load 还有一个列表框。这两个控件我们在主窗口中都用过，不过这次是根据资源脚本来创建和定位对话框。不仅如此，对话框还必须对按下回车键和用鼠标点击 OK 按钮作出相同的响应。这里不必对对话框中的编辑控件进行子类操作。现在，你也应该明白我为什么选择常量 IDOK 作为主窗口中子类编辑控件的命令标识符。

关于对话框还有一问题——焦点。在主窗口中关于焦点的规则很简单——焦点总位于输入编辑控件上。但是，当存在多个潜在的活动控件时，就没有这么简单了。假设你作为一个用户激活了某个控件（例如通过输入信息激活对话框的多个编辑控件之一）。然后，你想把焦点暂时转移到另外一个应用程序中，等处理完此程序中的事情之后再回到原来的程序中完成

输入。你肯定希望焦点回到那个你要进行输入的编辑控件上。这就要求对话框记住在焦点转移之前哪个控件处于激活状态，并在重新激活此控件时恢复焦点。显然这一切由对话框来完成。相反，如果是在其他任意窗口（非对话框）使用了其二控件，你就必须在它的控制器中添加特殊的代码来正确处理焦点恢复问题。

另外，对话框还提供使用键盘在不同控件之间切换的方法。你可以通过敲击 Tab 键把焦点在控件组之间来回切换，也可以通过方向键在控件组（例如单选按钮）之中切换。

你也许会问，为什么不把计算器的顶层窗口也做成对话框呢？这样可以立即解决若干个问题。确实有一些应用程序使用对话框作为主窗口——Windows 系统自带的计算器小程序就是这样。但我想编写的是一个主窗口包含菜单、状态栏以及窗口大小可变的主流 Windows 程序。归结到一点，把一堆控件放在一个大小可变的父窗口中更简单。

对话框提供了处理所拥有控件的一般性方式。但是，如果你想希望对话框做些其他有用的事情，你就必须拦截控件和对话框之间的通信消息。例如，至少你可以在用户点击 OK 按钮（或者 Cancel 按钮）时关闭对话框。如果用户通过对话框输入了一些数据，你可能需要从（一个或多个）控件中读取它们并传递给调用者。也许你还想在多个控件中进行协调。所有这些都可以通过对话框过程来实现。

对话框过程是窗口过程的简化版本。对话框从控件收到的所有消息首先由这个用户自定义过程进行过滤。对话框过程和窗口过程的主要区别在于，对话框过程不调用 DefWindowProc，而是在不处理某条特定消息时返回 FALSE。另外一方面，如果它处理了某条消息，将返回 TRUE。注意这里的 TRUE 和 FALSE 是 Windows 定义的布尔值，不是 C++ 中的 true 和 false。

> Windows 的布尔值是整数的类型定义。FALSE 被定义为 0（使用#define 宏），TRUE 是被定义为 1。TRUE 的定义和 C++中的定义发生冲突。当 C++中 true 转换为整数时，可以是任何非 0 的值。不幸的是，Windows 经常把布尔值直接和 TRUE 进行比较，所以如果布尔变量返回非 0 但不是 1 的话，程序就会出问题。

这就要求我们必须找到一种封装对话框的方式，使它的使用更简单、更不容易出错。

对话框和函数一样都是通过 UI 来获取数据。你可以用一些初始数据（参数）“调用”对话框，并返回一个结果，就好像是从用户得到的数据一样。

> 我们现在所讲的是所谓的模式对话框。在模式对话框的执行过程中，父窗口被暂时挂起——它不响应任何用户输入（不过还是可以刷新它自己）。只有当用户关闭了模式对话框之后，他才能继续和父窗口（一般是顶层应用程序窗口）交互。

我们总可以把输入和输出数据封装到某些用户自定义的数据结构中。但我们还要定义活动部分，即可以插入到一般对话框过程中的控制器。我们甚至还可以组合输入/输出数据与这一控制器，这样，它就可以响应控制消息对数据进行操作。

下面是一个关于如何使用这个抽象的概念来显示 About 对话框的简单例子（我们以自顶向下的方式设计此对话框）。

```
AboutCtrl ctrl;
Dialog::Modal dialog(_h, IDD_ABOUT, ctrl);
```

对话框对象有三个参数：拥有者窗口（这里是顶层窗口）、对话框 ID（用来标识资源中相应的脚本文件）和控制器对象。

控制器由库类 Dialog::ModalController 派生而来。在这个 About 对话框的最简实现中，它只覆盖了 OnCommand 方法以拦截 OK 按钮的动作。

```
bool AboutCtrl::OnCommand(int ctrlId, int notifyCode)
                  throw(Win::Exception)
{
  if(ctrlId == IDOK)
  {
    EndOk();
    return true;
  }
  return false;
}
```

EndOk 方法终止对话框，并返回调用者成功代码。

我还引入了一个新的针对 Windows 的异常类 Win::Exception。大多数 Windows API 函数都不会返回详细的错误代码。相反，调用者应当在函数执行失败后立即调用 GetLastError 获取错误代码。这正是 Win::Exception 构造函数的作用。然后调用 FormatMessage API 把错误代码翻译成文本信息。



Script 16

Save 对话框还有其他一些功能。它包含一个“_path”字符串，可以存储对话框返回的路径。它还有一个 Win::Edit 对象，这个对象可以和对话框中的编辑控件进行通信。OK 处理器从编辑控件中读取字符串并拷贝给 _path。CANCEL 处理器只是通过调用 Dialog::ModalController::EndCancel 方法终止对话框。

```
bool SaveCtrl::OnCommand(int ctrlId, int notifyCode)
                  throw(Win::Exception)
{
  if(ctrlId == IDOK)
  {
    SetPath(_edit.GetText());
    EndOk();
    return true;
  }
  else if(ctrlId == IDCANCEL)
```

```
    {
      EndCancel();
      return true;
    }
    return false;
  }
```

调用者通过调用 IsOk 方法区分成功输入（激活 OK）和无效输入（激活 Cancel）。下面是对应的顶层代码：

```
SaveCtrl ctrl;
Dialog::Modal dialog(_h, IDD_SAVE, ctrl);
if(dialog.IsOk())
{
  std::string const & path = ctrl.GetPath();
  Serializer out(path);
  _calc.Serialize(out);
}
```

控制器必须提供对话框窗口句柄（对话框是编辑控件的父窗口）和控件 ID 来初始化_edit 对象。这个初始化在 OnInitDialog 方法内部进行。

```
void SaveCtrl::OnInitDialog()throw(Win::Exception)
{
  _edit.Init(GetWindow(), IDC_EDIT);
}
```

注意这里使用的 Win::Edit 类和前面封装自身编辑控件的父窗口所使用的 Win::Edit 类是一样的。唯一的区别是我们不必使用生成器来创建编辑控件——对话框本身可以根据资源脚本创建控件。我们可以通过在 Win::Edit::Init 内调用::GetDlgItem 获得控件窗口的句柄。

Script 17

Load 对话框的功能更先进。它包含一个列表框，我们用它来显示当前目录中的文件列表。它还有一个静态文本控件，列表框用来显示当前目录的路径。Win::ListBox 的 ListDirectory 方法负责目录列表以及静态文本控件的初始化。

```
void LoadCtrl::OnInitDialog()throw(Win::Exception)
{
  _listBox.Init(GetWindow(), IDC_LIST);
  _listBox.ListDirectory(GetWindow(),
                         GetBuffer(), IDC_STATICPATH);
}
```

当用户单击 OK 按钮或者双击某一项时，对话框读取列表框中被选中条目的完整路径。GetSelectedPath 方法用数据填充缓存器，如果选中的是一个目录（不是文件）就返回 true。

如果是一个目录，我们就改变当前目录并重新初始化列表框。如果是一个文件，就关闭对话框，让调用者从缓存器中读取文件路径。

```
bool LoadCtrl::OnCommand(int ctrlId, int notifyCode)
                throw(Win::Exception)
{
  if(ctrlId == IDOK
   || ctrlId == IDC_LIST && notifyCode == LBN_DBLCLK)
  {
    if(_listBox.GetSelectedPath(GetWindow(),
                                GetBuffer(), GetBufLen()))
    {
      // 选择目录
      ChangeDirectory();
    }
    else if(_listBox.IsSelection())
      EndOk();
    else
      EndCancel();
    return true;
  }
  else if(ctrlId == IDCANCEL)
  {
    EndCancel();
    return true;
  }
  return false;
}
```

下面是 ChangeDirectory 的实现方式：

```
void LoadCtrl::ChangeDirectory()
{
  ::SetCurrentDirectory(GetBuffer());
  ResetBuf();
  _listBox.ListDirectory(GetWindow(), GetBuffer(),
                         IDC_STATICPATH);
}
```

我承认这种读取文件的用户界面有些过时。第一，它不能显示长文件名，实际上，它甚至不能显示 NTFS（Windows NT 所使用的文件系统）中的文件列表。其二，在不同的目录中浏览文件还不是十分直观。有一个 Windows API 函数 GetOpenFileName 可以实现 Load 对话框的所有功能，并且它的用户界面要好得多（见下面的练习 5）。我使用这种老式的界面（在一些应用程序中还是使用这种界面）只是为了演示包含特殊控件的对话框的用法。

Script 18

最后的修改（修饰性的修改）是在一个更小的窗口中启动计算器。这由一个新的方法 Win::Maker 完成。

练习

1. 当用户在历史记录中双击某一项时，把选定的字符串拷贝到编辑控件中，这样用户就可以编辑和重新执行它。

2. 一个真正的动态菜单系统应该还可以根据应用程序的状态删除某些特定的菜单项。改变菜单系统，清除现有的弹出菜单，当用户每次点击菜单栏上的菜单项时都从头开始创建一个新的弹出菜单。

提示：给 Menu::Manip 增加 AddItem、AddSeparator 和 Clear 方法。如果它们在弹出菜单中不出现的话，Commander 的“can”方法应该返回 Cmd::Invisible。

3. 在菜单栏中增加“Function”项。对应的弹出菜单显示所有内置函数的列表。当用户选择某个内置函数时，它的名字和括弧应该添加到编辑控件中字符串的后面。

提示：这个弹出菜单的初始化不应该是静态的。它应该使用计算器的函数表进行初始化。

4. 给 Load 和 Save 命令分别增加加速键 Ctrl+L 和 Ctrl+S。使用一个静态初始化的加速键表，把这个表和命令矢量（用于从命令名到命令 ID 的转换）一起传递给加速键生成器。创建加速键表的 API 函数叫做 CreateAcceleratorTable。加速键表是一种资源（通过 DestroyAcceleratorTable 释放），所以，在设计你的类时要应用资源管理。

为了添加加速键，需要把一个消息预处理器的引用从 WinMain 传递到 TopController。当加速键创建完成后，调用 MsgPrepro::SetKbdAccelerator 方法激活它。

提示：记住改变菜单项的显示字符串，把加速键包含进去。例如 Load 菜单项应该显示为“&Load...\tCtrl+L”（tab 符标标记“\t”表示右对齐加速键字符串）。

5. 把 Load 命令浏览目录的功能转由使用 GetOpenFileName 函数实现。

6. 也许你认为通过直截了当的通知方法比利用通知接收器方法更容易实现。那么请读者来试一试。

让控制器在解析器每处理完一个新的表达式之后就调用一次 View::RefreshMemory。你也许能回忆起来，这个处理过程是对编辑控件检测到回车键后发送 IDOK 命令的响应。

为了避免太强制化（也为了避免屏幕不停地闪烁），对内存显示进行逐行更新，并且只修改那些自最后一次刷新以来确实改变了的项。这意味着你必须采用某种机制标记（取消标记）单个内存项。你可以在你的代码中应用以下程序片段：

```
void View::UpdateMemory()
{
  int count = _memoryView.GetCount();
  for(int i = 0; i < count; ++i)
```

```
    {
      int id = _memoryView.GetData(i);
      if(_calc.HasValueChanged(id))
      {
        _calc.ResetChange(id);
        std::string varStr = FormatMemoryString(id);
        _memoryView.ReplaceString(i, varStr);
        _memoryView.SetData(i, id);
      }
      }
      int iNew;
      while((iNew = _calc.FindNewVariable())!=
                                    SymbolTable::idNotFound)
      {
        _calc.ResetChange(iNew);
        std::string varStr = FormatMemoryString(iNew);
        int i = _memoryView.AddString(varStr);
        _memoryView.SetData(i, iNew);
      }
    }
```

第四部分

知识扩展

前面所讲述的符号计算器只是一个小软件项目，可以由单个程序员完成，所花的时间也会太多。软件工程真正的挑战是构建复杂度比这个项目高几个数量级的程序。到目前为止，我所介绍的一些编程技术，既可应用于小型软件项目，也可应用于大型软件项目中。不过，对软件工程来说，除了这些一般的编程技巧外，还需要掌握其他很多知识。

编程就是一个化繁为简的过程。如果一个问题的复杂程度使得单个程序员无法解决，那么就由一个小组来解决。但是一个包含 n 名程序员的开发小组的能力和工作方式并不直接等同于单个程序员（他的智慧 n 倍于一般程序员）的工作能力和方式。团队工作一般都是把一个程序分解成很多易于管理的小块，由单个小组成员完成不同的子块，最后再把它们组合在一起构成最终程序。这项工作不仅需要程序开发的具体知识，而且还需要队员间的协作技巧。

第19章 关于软件

我们现在抛开编程的技术细节，概览整个软件开发过程。

19.1 复杂性

复杂性处理、人类智慧的有限能力、分解和征服、抽象。

软件工程的实质就是将复杂问题化繁为简。这也是软件工程中要求最高的部分，它不仅需要科学性，还需要创造性。为什么我们需要用特殊的方法来处理复杂问题呢？答案在于我们的大脑能力。我们大脑的快速反应能力只能处理很少数量的对象——不管它是什么类型的对象，思想、图像、或者文字。至于能处理的对象的具体数目，由对象的复杂程度决定。这不是我杜撰的。George Miller 在 1953 年的 *The Psychological Review* 杂志上，发表了一篇名为“The Magical Number Seven Plus or Minus Two”的学术论文，他在文章中认为，人类在理解事物方面有着天生的局限性。

本质上，我们人类有两条途径处理复杂问题，分解—征服方法和抽象方法。分解—征服方法的基本原理是，把复杂问题看成一个树状结构。树的每个节点只有少量的树枝，树枝的数目在人类可以处理的能力范围之内。从树叶到根部或者从根部到树叶的遍历需要呈对数的总步数，同样假定也在人类可以处理的能力范围之内。例如将科学知识体分为人文科学和自然科学（两个分支）。自然科学又分为不同的领域，计算机科学就是其中的一个领域。

计算机科学专业的学生要理解 Kernighan 和 Ritchie 编著的 *The C Programming Language* 一书并不需要多少人文科学知识。另一方面，诗人不懂 C 语言编程也能赋诗。人类知识的树状划分不仅有利于深入交流和探索，而且允许不同团队间进行分工协作。我们可以把整个人类比作一个巨大的团队，正在从事一项了解世界的庞大项目。

另一个由所有生物体共有、由人类完美运用的强大工具是抽象。“抽象”这个词和“减少”有着相同的词源。抽象的意思就是把一些不重要的特点去掉。例如要识别一辆

汽车，请想象一下我们可以去掉多少无关紧要的特征。你可以去掉喷漆的颜色、牌照、雨刷、载重等等。当一只鸟在形成“捕食者”概念的时候，它也不自觉地在进行抽象。抽象并不是 100%的准确：例如稻草人可以吓走乌鸦，因为稻草人符合乌鸦心目中的“捕食者”形象。

分解和抽象可以结合在一起使用，一般我们称它们为分解—抽象模式。一个复杂的系统可以形象地比作一个有很多互联节点的巨大网络（如图 19-1 所示）。我们把这个网络分解成几个“对象”——节点的子集。一个好的分解应该使得对象内部的连接尽可能的少（如图 19-2 所示）。为了描述分解出来的对象，我们再使用抽象。尤其是我们可以用对象之间的连接方式（接口）来描述它。我们可以通过尽量减去不重要的特点来简化它的内部结构（如图 19-3 所示）。在分解的每一步，整个系统都可以通过几个成功抽象对象之间的交互关系来理解。如果无法实现这一目的，那我们应该放弃这一途径。世界的真正神奇之处在于，世界的大部分（也许是全部）都可以用这种模式来实现。

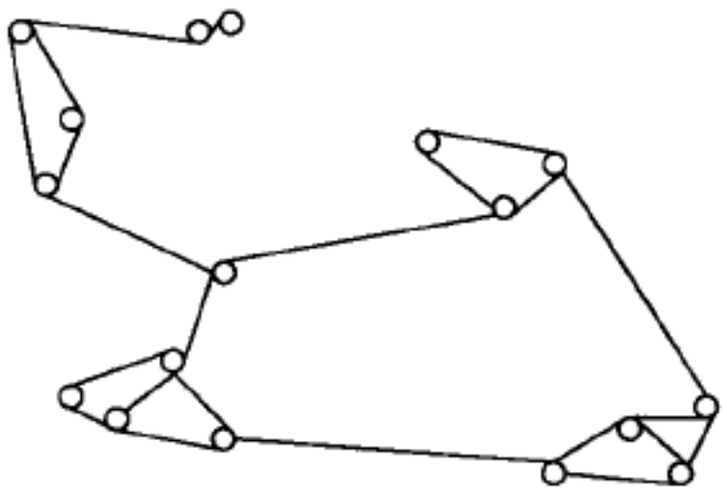


图 19-1　一个复杂系统

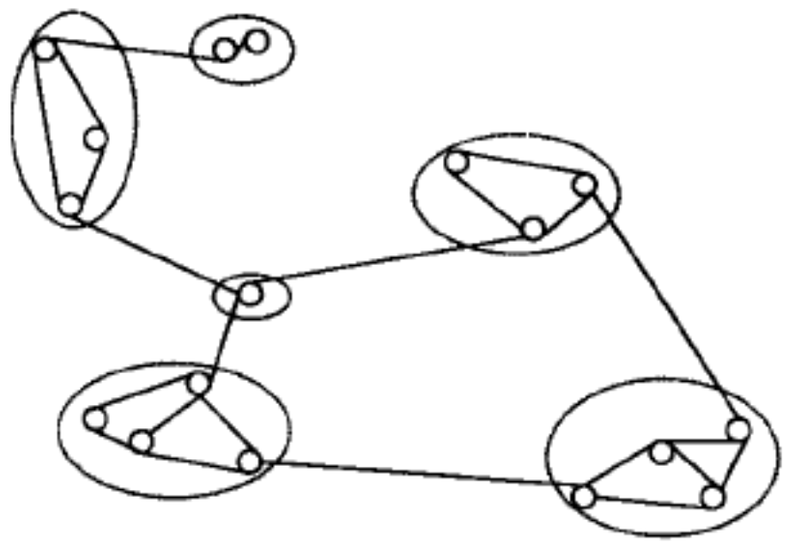


图 19-2　从复杂系统抽象对象

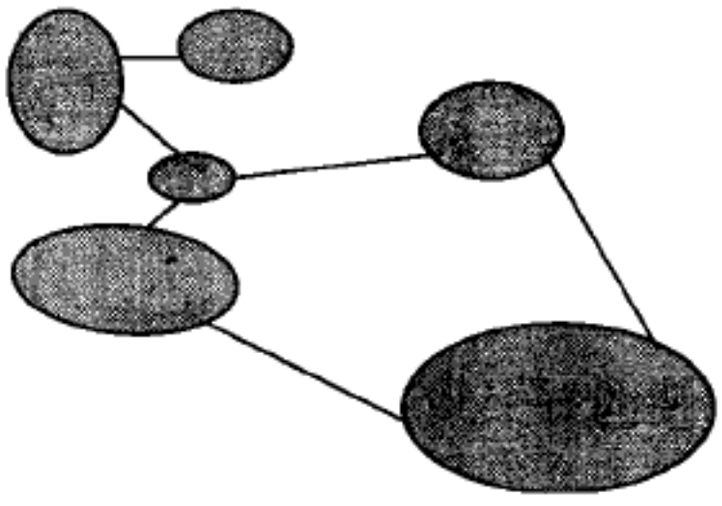


图 19-3　抽象对象后复杂系统的高层视图

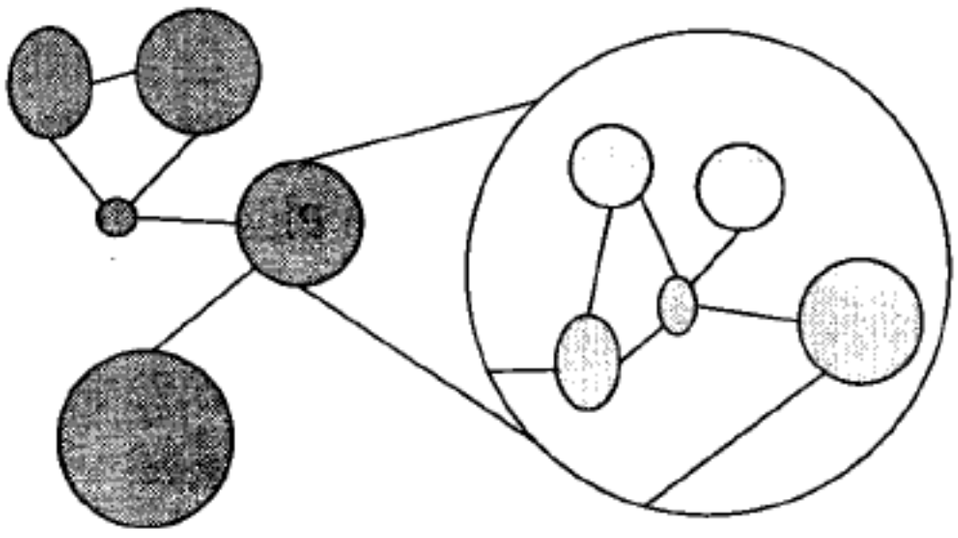


图 19-4　一个复杂系统的分形结构

分解—抽象过程在把对象分解成子对象的过程中不停地递归重复。为了进入一个更深的层次，把所有被减去的特点加回来，取消对对象所作的抽象，把对象分解成子对象，然后使用新的抽象来定义它们（如图 19-4 所示）。对象应该可以通过几个成功抽象的子物体来理解。这个递归过程在一定情况下可以创建一个自相似、分形状的结构。

在软件工程中，我们把一个大项目分解成易于管理的小块。我们使用抽象的方法定义、命名和描述这些子块。就像我们讨论的符号表、解析器、索引、存储层等一样，它们都是抽象。我们通过抽象把一个大问题分解成若干个小问题。

19.2 软件的分形本质

首先，我用本书中大家已经很熟悉的的软件项目——计算器作为例子来说明这些概念。这个项目的顶层是由一组相关的对象构成（如图 19-5 所示）。

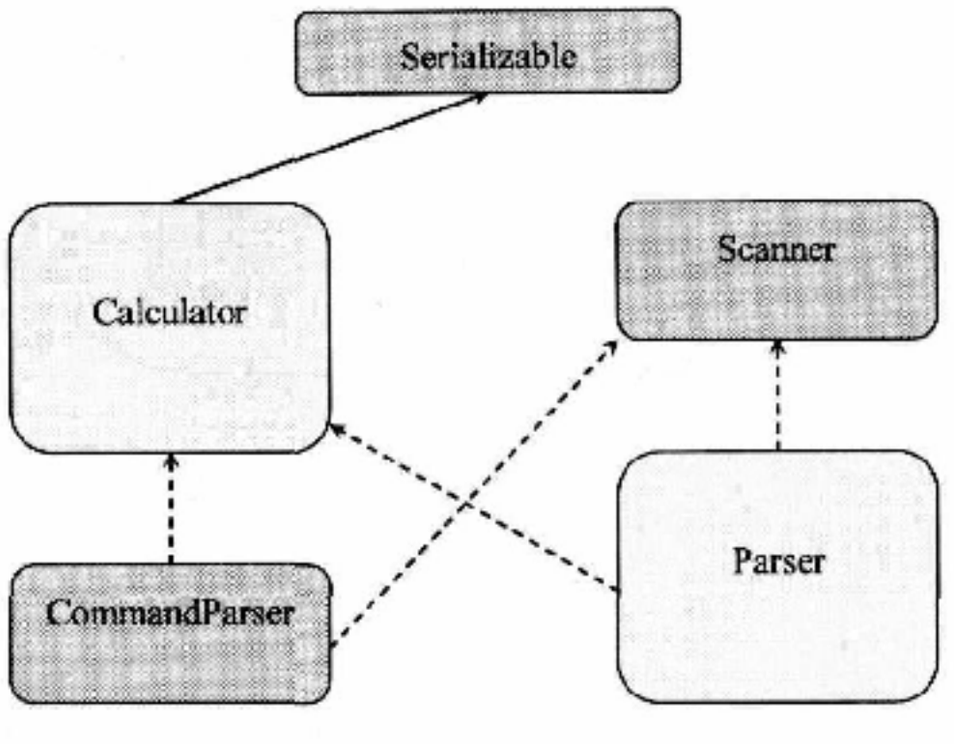


图 19-5 计算器对象的顶层视图

只使用这些对象（通过它们的公共接口和功能），我们就可以解释清楚程序是如何工作的（main 函数是干什么的），从这个意义上来说，这个系统是完整的。我们不必知道对象是如何执行其功能的，只要知道它是干什么的就行了。

那么程序是如何工作的呢？首先，在 main 中创建 Calculator。Calculator 是可串行化的，也就是说，它的状态可以存储和恢复。注意，在这个层次中，我们不必知道流的任何知识——它们是没有可见接口的黑匣子（这也是我没有把它们画进图中的原因）。

一旦创建好了 Calculator，我们就进入了一个循环，在此循环中，我们从用户处获取文本流，并用它创建 Scanner。Scanner 判断用户的输入是不是命令。如果是命令，我们就创建一个 CommandParser；如果不是命令，我们就创建一个 Parser。两者都需要访问 Calculator 和 Scanner。CommandParser 可以执行（Execute 方法）一条命令，而 Parser 可以解析（通过 Parse）输入，计算（通过 Calculate）结果。然后我们显示计算结果并返回到循环的起始处。当 CommandParser 从 Execute 方法中返回 stQuit 状态时，循环终止。

这就是整个工作过程！没法比这更简单了。然而，第一次尝试抽象能达到这样的水平并不容易。实际上，我们的第一次抽象也并不完美!为了获得这一简单的结构，我们必须进行一系列的改写。本书第二部分描述的所有技巧和规则都为实现这一目标而服务。

下面继续我们的学习。让我们放大其中的一个顶层组件——计算器。同样，它也可以用一组相关的对象来描述（如图 19-6 所示）。

我们仍然可以只使用这些对象（以及几个上层对象）来解释所有计算器方法的实现。

接下来放大 Store 对象，可以得到非常相似的图示（如图 19-7 所示）。

我们可以依此继续下去，就像在那些 Mandelbrot 程序中一样，你可以把图中任何部分放大来看，你会发现哪些相同、哪些不同。通过数学分形，你可以无限深入下去，同时可以看到相同的无限层细节。对于一个软件项目来说，最终到达的层次是纯内置类型和命令（当然，你还可以继续分形到汇编语言、机器微码、门控、晶体管、原子、夸克、超弦等层次都是可以的，不过这超过了本书的讨论范围）。

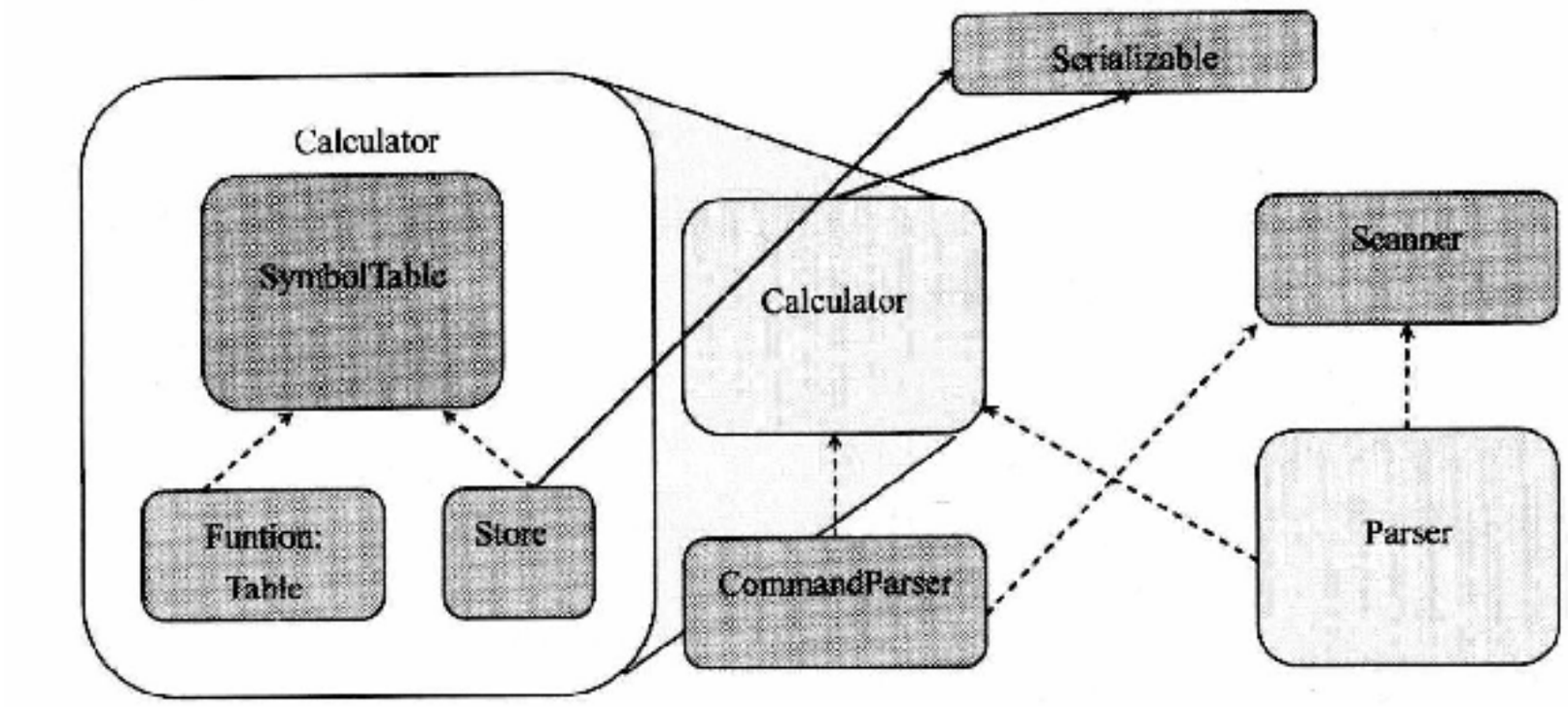


图 19-6　Calculator 的放大结果

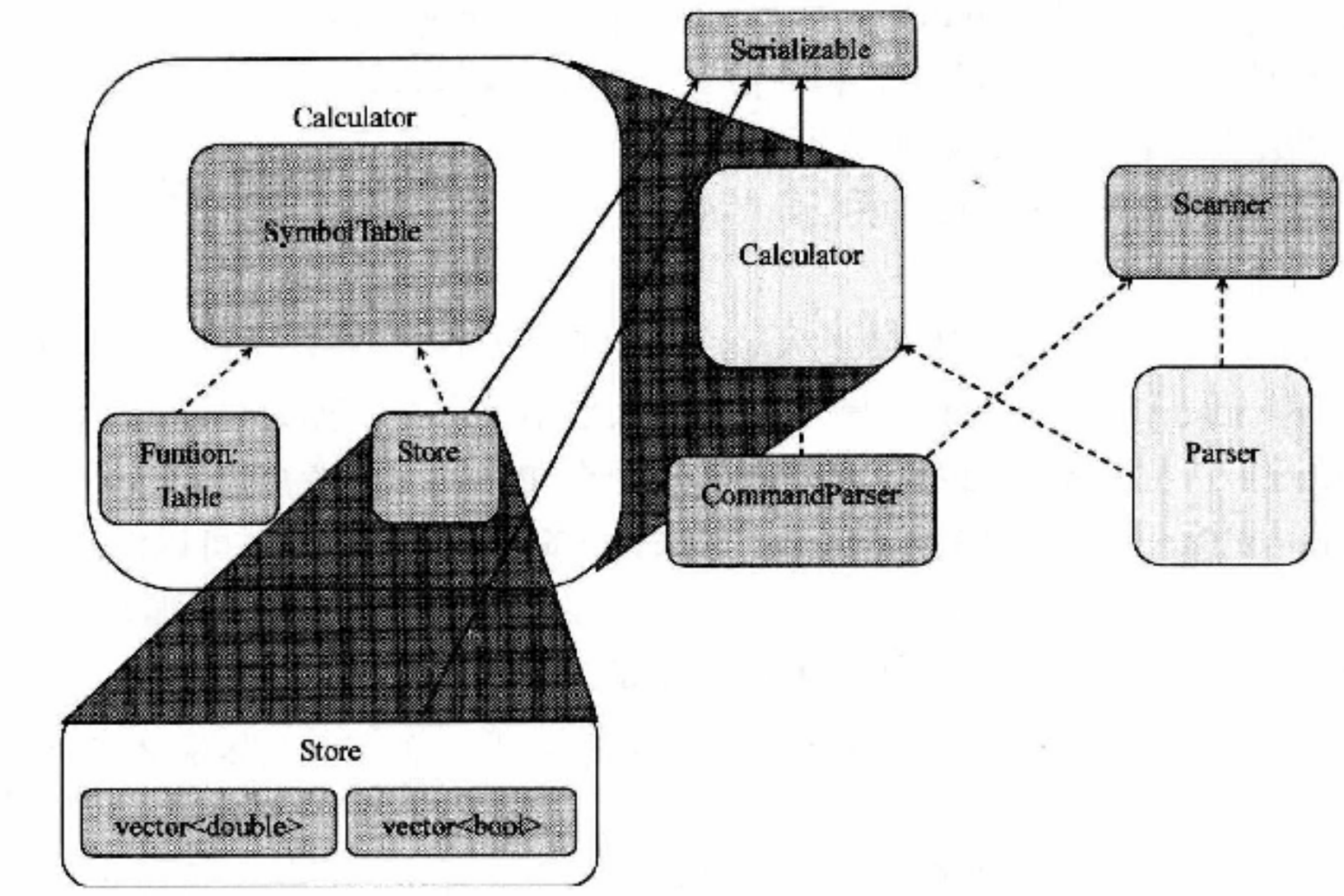


图 19-7　Store 对象的放大结果

19.3　项目的灵活性

项目的生命周期、编程的周期性、不同的阶段、末端开放设计、与生物体类似的程序。

每个软件项目都有开始，很少有结束（除非它们被管理人员取消了）。你必须习惯这种末端开放的开发方式。这样可以为你和你的合作者减轻痛苦。假设从一开始

- 将添加新特点
- 部分程序将重写

- 你的代码将由其他人来阅读、理解和修改
- 将会有 2.0 版（及更高升级版）

为版本 2 而设计、为版本 1 而实现。一些版本 2 中的功能在版本 1 中应该哑组件处理。

软件项目开发包含不同等级的周期。最长的周期是它的主版本周期。在这期间一般还有一个或几个次要的版本周期。主要版本的开发经历以下阶段：

- 需求（或外部）规范
- 架构设计（或重新设计）
- 实现
- 测试和错误修改

这些阶段是互相交错的。架构设计的结果可以反馈给需求规范阶段。可能实现有些功能代价太高；或者在设计中又产生了新的需求。

主要的反馈是从实现阶段反馈到设计阶段。甚至有人建议软件开发应该经历抛弃设计原型并重新设计这样一个实现周期。抛弃设计原型通常是对开发时间的巨大浪费。它意味着在设计和研究问题上花的时间太少了，这种实现方法不合适。

与设计和实现最终产品相反，程序员在设计和实现原型、不同的结构块时不应当使用不同的方法。不遵守这个规则是程序不一致的表现。这不仅会影响士气，而且还不能节省开发时间。正相反，我和同事不同的方法编写“脚手架”代码，我们被代码中的错误和遗漏弄得筋疲力尽，大量的时间浪费在排错上，所以最终我们决定采用同样的方 法编写最终产品代码。这样做的另一个效果就是，只要“脚手架”代码通过实现周期并应用到最终的产品中（你肯定会惊诧这种事情发生的频繁性），它就不会导致任何大的灾难。

回到实现周期中：任何主要组件的实现或者重写都需要事先进行仔细、全面的设计或重设计。在这个过程中，要经常更新文档，及时在架构规范中加入新设计的部分。设计工作通常都是末端开放的，它总是受到实现方案的强烈影响。这就是为什么要经常更新文档的原因。和项目不同步的文档是没有用的（甚至更糟糕的是，还会导致错误信息）。

实现本身也是在一些小周期中完成的，这些周期就是每个程序员都熟知的基本编辑—编译—运行周期。注意，测试是如何与开发交织在一起的。周期中的运行部分可以作为简单的完整性测试。

在这个层次上，程序员的工作类似于内科医生的工作。与替病人看病一样，首要原则就是不要伤害病人。在编程中，这一原则叫做“不要肢解代码”。程序应该被当做器官来看待。你必须保持它的活性。扼杀程序再让它复活是不正确的开发方式。将所有的改动分成一些独立的小步骤来进行，并尽可能多进行测试。在进行大型“器官移植”的时候，也许有些功能暂时失效，但一般来说，程序应该随时保持全部功能。

最后，提醒你一些开发中的禁忌，即一些要避免的项目开发方式（以及在很多地方它们如何仍然被采用的）。不要太快跳跃到实现阶段。要有耐心，要顶住经理要你尽快拿出一个演示版本的压力。在写代码之前要酝酿再三。当你想法模糊，只是企图通过尝

试和调试来得到最终代码，那你暂时不要坐在计算机前开始编程。不要编写拖泥带水的代码而指望“事后再清理 ”。与其编写冗余代码，还不如干脆舍弃某些功能，两者间的区别相当大。

19.4　程序员的灵活性

谦虚、直率、团队精神、奉献精神。

程序员是人，如果不认识到这一点会引起很多误解。虽然程序员的很多时间都是在和计算机打交道，但这不意味他们就和其他人不一样了。归根到底，计算机是为人服务的，而不是人为计算机服务，所以编程工作必须以人为本。这听起来道理浅显，但在现实生活中，这条简单的规则却经常被违反。例如强迫程序员用汇编语言（或者 C 语言）进行编程就是一个与之背道而驰的例子。围绕低层数据结构如哈希表、链表等构思设计是也是如此。

计算机还不能完成自我编程（实际是正好与之相反），这一事实表明我们人类具有自己的优势；而某些人类的工作可以由计算机完成的事实却表明计算机同样具有自己的优势。软件工程的基本等式为：

```
人的创造性+计算机的速度和可靠性=程序
```

试图把人的速度和可靠性与计算机的创造性结合起来编写程序是一个大错误。我们一定要面对这样一个事实：我们人类的速度和可靠性比不上计算机。计算机的速度很快，所以当计算机在编译代码时，还需要程序员等待，那肯定是什么地方发生了错误。另外，若认为程序员不通过任何编译器、连接器或调试器的帮助就可以写出没有任何错误的代码，那也是不可能的。如果认为程序员甩开了笔和纸，就可以通过反复地尝试和调试找到没有一般保护错误的参数组合，那他就更是大错特错了。

优秀程序员所具备的品质与武术信徒所具备的品质相似，一方面要谦虚、耐心、直率；另一方面要具有团队精神和奉献精神。最重要的是，不要相信任何人包括自己。

- 谦虚：认识到自己的缺点。作为程序员，不可能写出没有错误的代码，是人就要犯错误。编写程序的时候就应该预备错误的出现。使用一切可用的手段防止错误的发生。不要吝啬使用断言。使用堆检查。花点时间增加调试输出。
- 耐心：不要盲目而动。要耐心地构建坚实的基础。在写代码之前做好设计工作。写出下一代人都能用的健全代码。
- 简单：去掉不必要的代码。如果你发现了一个更简单的方案，那么应用它重写程序中的相应部分。每个程序都可以简化。避免特殊情况的出现。
- 奉献：编程不是从早上九点到下午五点的按部就班式工作。当然，我也不是说你晚上和周末都必须工作。如果是这样，那说明你的安排不合理。不过你应该活到老学到老，紧跟时代前进的脚步。如果你不去不断地充实自己，你就会被日新月异的技术抛在后面。

- 团队精神：程序员单独开发项目的“个人英雄”时代已经一去不复返了。你必须和其他一起协作来完成任务。你必须具备一定的交流技巧，你要遵循一定的标准、编码约定、注释约定等等。当某些约定已经丢失了意义时，要随时和团队中的其他成员对它们进行讨论并修改。有些人鼓吹“不合理的约定也比没有约定好”，不要轻信他们的话。
- 怀疑论：不要相信任何人的代码，甚至连自己的代码也不要相信。

第20章 设计策略

对于一个软件项目来说，在写下第一行代码之前具体工作早已开始了。如果没有良好的设计，就不可能编写出一个商业级的应用软件。和其他产业不一样，软件开发人员应该积极参与产品设计。

20.1 自顶向下的面向对象设计

顶层对象、抽象和比喻、组件。

从哈希表、链表、队列、树这样的对象开始设计，并把它们结合在一起是很容易的。不过应该避免使用这样的自底向上的实现驱动方式。自底向上设计的程序结构比较混乱，程序就好像一锅汤，各种对象就像蔬菜、肉片、各种面条一样漂浮在汤上面。这样的程序看起来像是面向对象的——它有面条、蔬菜、肉片等“类”。但是，因为你几乎不用改变链表、队列、树等的实现方式，所以它们的面向对象性优势没有发挥出来。你必须花大部分时间对这个无形的汤进行维护和修改。

另一方面，当使用自顶向下方法时，你就把程序分解成少量的相互作用高层对象。指导思想就是把要处理的对象（类）类型的数目限定在人脑可以处理的范围之内。顶层对象又分为两组：程序的主要对象以及联络各主要对象的通信对象。如果程序是交互式的，你的设计应该从用户界面以及处理用户输入和屏幕（电传）输出的对象开始。

一旦确定了顶层对象，就要预演对象之间的交互作用（有时也叫做实践检验）。首先是初始化过程，你要决定哪些对象需要首先创建以及它们的初始状态是什么。除了可能在顶层之外，在其他任何层次都要避免使用全局对象。当所有的对象初始化完成以后，假设你自己是用户，测试一下这些对象会对用户输入做出什么样的反应、它们的状态如何改变以及它们交换什么样的通信对象。你应该能在顶层描述这些相互作用，而不必知道底层实现的细节。

经过上面的演习，你应该对顶层对象的接口以及它们之间要达成的契约做到心中有数（也

就是说，应该知道某个调用所需的参数以及得到的结果）。每个对象的定义应尽可能简洁明了，它们的功能应该构成一个连贯、完备的抽象。在说明文档中描述对象及其相互作用时，应尽量使用常规语言而不用代码。请时刻记住，软件项目要以人为中心，而不是以计算机为中心。如果你能用常规语言很好地描述某个对象，那么通常这一抽象就是成功的。

对那些不易抽象的东西使用比喻。你可以把一个编辑器比喻为一张纸，一个日历比作一个日程表，画笔、画刷、橡皮擦和调色板比作画图程序等。在用户界面的设计中要不断地用到比喻，并且在其他层次设计中也经常用到。文件、流、信号量、端口、虚拟内存页、树、堆栈等等——所有这些都是很有用的底层比喻的例子。

正确进行抽象总是很重要，并且在大型项目中尤其至关重要，因为在大项目中顶层对象是由不同的小组实现的。这些对象叫做组件。当整个项目全面启动之后，任何对组件接口和契约的改变都会带来灾难性后果。根据使用这一特定接口的组件数量，后果的严重性或轻或重。灾难的严重性大概可以用“里氏震级”来衡量。每个项目都会经历几次这样的“地震”——这就是软件开发!

接下来对每个顶层对象重复同样的设计步骤。首先，把对象分成若干意义明确、接口完备的个子对象。如果有必要，再往下进一步细分子对象，直到获得足够详细（也不要太详细）的设计。在实现不同的子块过程中，不断重复以上过程。最终目标就是得到项目的自相似、分形结构。整个程序的顶层描述应该和每个组件及其子组件、对象及其子对象等的描述类似。不管放大还是缩小来看，你都应该看到或多或少相同类型的图示，各自包含一些独立的对象来实现某些特有的功能。

20.2 用户界面

用户界面设计、输入驱动程序、模型—视图—控制器模式。

现在，即使是最简单的程序也提供了一些交互性。当然，UNIX 模式的程序现在仍然能见到，这些程序从标准输入设备接收一维字符流，另外分配一个流作为标准输出。但是随着图形用户界面（GUI）程序的出现，这种所谓的“命令行界面”很快就被淘汰了。对于用户来说，图形界面意味着更友好、更自然；对于程序员来说，这意味着观念的转变和更多的付出。

借助于操作系统和一些合适的工具，设计和实现用户界面并不是很困难，至少对那些不需要图形化的程序来说是这样的。真正的重点在于改变观念。交互式程序的绝大部分是输入驱动的。程序根据用户的输入执行响应动作。在最高层，一个交互式程序可以看作是针对外部产生事件的一系列事件处理器。每一次键盘敲击和鼠标点击都要进行相应的处理。

面向对象设计方法应付交互性挑战的手段就是第 16 章所讲述的模型—视图—控制器范例。控制器对象是所有外部事件（有时包含内部事件）的焦点。它的功能是尽可能清楚地翻译出这些事件的含义，然后决定用程序中的哪个对象对它进行处理。然后把相应的消息发送给这一对象（这是 Smalltalk 中的做法；在 C++中，我们只需要调用相应的方法就行了）。

视图负责程序的可视化输出。它把其他对象的请求翻译成图形化表示形式并显示。换句话说，它对输出进行抽象。画线、填充区域、编写文本和显示光标等都由视图负责完成。

把输入集中在控制器、输出集中在视图可以使程序的其他部分独立于复杂的输入/输出系统（也可以使程序易于移植到其他图形功能和接口稍有不同的环境下）。和输入/输出细节无关的程序部分叫做模型，它是程序最艰难的部分，也是程序的大脑。在简单程序中，模型对应于单个对象，不过通常它都是一组顶层对象的集合。模型的不同部分由控制器根据外部事件来激活。当状态改变时，模型在它认为合适的时候更新视图。

作为自顶向下设计交互式程序的一条规则，你应该首先建立控制器和视图的功能。任何在用户动作之前发生的事情都可以看作是这些组件和模型本身的初始化。M-V-C 三元组合也可能出现在程序的低层，用于处理特定的控件、对话框、编辑控件等。

20.3　需求规范

目的陈述、功能、用户界面、输入、输出、大小限制和性能目标、特点、兼容性。

在做项目的其他工作之前，第一个要写的项目文档是需求规范（也叫做外部规范）。在大型项目中，需求规范通常都是由一组专门研究市场、负责用户反馈、用户测试等工作的人员来编写。如果程序专门为某外部客户开发，那么需求规范就应该由此客户准备。不过，不管由谁编写需求规范，结构设计、实现人员与规范编写组之间应当建立良好的反馈机制。

需求规范中最重要的部分是目的陈述——也就是这个软件系统的用途。有时，重新审视程序的开发目的，可能会带来设计上的一些新见解以及重点的转移。例如，把编译器定位为一个检查源文件错误，并偶尔生成目标文件（没有错误时）的程序，就有可能开发出一款非常优秀的产品。

目的陈述中也可以包含程序所使用的主要比喻的讨论。例如，一个编辑器可以描述为（比喻成）一个处理文本行的工具。不过，根据经验来看，把编辑器比喻为一张纸更贴切一些。电子表格类软件的流行也是因为很好地使用了其他类型的比喻。

接着就是程序功能的详细描述。在一个字处理软件的需求规范中，可能有文本输入、段落格式、样式创建等方面的描述。在一个符号化操作程序中，则要定义被处理的不同类型符号对象及表达式，以及各种可能用到的转换等等。这一部分规范 应该告诉设计者和实现者要实现一些什么功能。哪些功能是必须实现的，哪些是可选的。

然后是用户界面和可视化比喻。这部分的改动是最频繁的。当创建第一个用户界面原型之后，它首先要经过开发人员，然后再是可能用户的严格测试，这种测试或多或少（大部分情况下不多）。有时候，经理可能不喜欢你设计的界面而要求你返工。界面的好坏关系着一个产品的成败，它的设计其实更多的是反映艺术造诣而不是纯科学技巧。

对用户界面设计的评论是仁者见仁、智者见智。每个人都可发表高见，也无需专门的培训。每个人都有不同的品味。设计界面的程序员可能是最没有资格来判断其好坏的人。他们

习惯于编程工具那种简单、神秘的界面，例如 UNIX 中的 grep、make、link、Emacs 或 vi 等命令。在任何情况下，用户界面的设计都是一件费力不讨好的事。

在用户界面后面是输入/输出规范。这部分规范描述程序接受什么样类型的输入，以及对应这些输入程序生成什么样的输出。例如，当用户点击了格式刷按钮之后，再点击文档中的某个字或者某个段落时，程序应该如何作出反应？或者，当程序读入一个包含用逗号分隔的数字列表文件时会发生什么？或者从剪贴板粘贴一副图片时会发生什么？

还可以定义速度和大小方面的要求。比如通常指定处理器种类、最小内存配置、最少磁盘容量等。当然，在不断增长的功能需求、保守的硬件要求以及出色的操作性能要求之间经常会发生冲突（根据我的经验，通常是首要保证功能方面的需求!但当项目进入攻坚阶段时，往往大量删减功能）。

最后是兼容性方面的需要。一个产品必须能理解（或转换）以前版本、同类产品生成的文件。在程序中包含一些兼容性方面的特性是明智的，它使将来的版本升级更加容易（必需版本戳）。

20.4 体系结构规范

顶层视图、关键数据结构和算法、主要的实现方法。

体系结构规范描述了程序的工作方式，以及为什么要这样工作。这个文档给出了开发者眼中的程序顶层视图。所有的顶层组件及其交互作用都进行一定程度的描述——这里的图表起着重要作用。文档应该能够清楚地表明，如果顶层组件根据规范实现了其功能，那么整个系统将能够正常工作。这样能大大减轻开发人员的负担——他们不必过多考虑各部分之间的依赖关系。另外，在文档中描述系统的理论基础也有益无害，或者至少列出一些参考文献。

体系结构规范中定义了主要数据结构，特别是那些永久数据结构。接下来主要是事件场景以及系统不同状态的描述。例如，程序可以是从零开始，也可以是在装入一些历史信息（文档、日志和永久数据结构等）后开始。程序可能还需要继续完成在前一个进程里中断的一些事务。它必须经过初始化，才可能进入稳定工作状态。

外部或内部事件有可能会引起数据结构转换和状态改变。所以还必须对新状态进行描述。在有些情况下，同时还要描述在这些过程中所使用的算法。要避免对实现方法进行过于详细的描述，因为它很快就会过时，对它进行维护也没有多大意义。

这里我们再一次提醒自己，编制文档是灵活多变的。它应当以一种易于更新的方式来编写。我们知道在实现周期中文档要修改很多遍，所以文档自身必须要有一个合理的结构。实际上，文档也应该是变化的，不断地更新文档、鼓励开发人员经常查阅规范文档有着重要的意义。

我们经常会碰到一些重要的对象，在体系结构规范中只有骨架，而在实现周期中，我们需要多次添加“血肉”。这时，我们可以扩充规范，或者就这个问题编写一个简短的单独文档。在这个单独文档中，我们可以描述重要的实现逻辑、算法、数据结构、编程注意点、约定等。

第21章 使用标准模板库

在大规模软件开发中，主要的挑战是如何高效组织众多的协作程序员。这不仅是组织上的问题，相互间的分工协作还反映在软件项目的高层设计中。

21.1 生产效率

信息爆炸、恶性循环。

大开发团队中程序员的工作方式为：

- 和其他程序员交流、参加会议、阅读文档、阅读和回复电子邮件。
- 等待其他人完成他们的工作、修改错误，实现一些重要的功能以及构建相应的组件。
- 调试、排错。
- 在计算机编译、装载一个大型程序、运行测试软件以及重新启动计算机时紧盯计算机屏幕。

最后，如果时间允许的话，还要

- 开发新的代码。

众所周知，在 n 个元素之间有 n（n-1）/2 可能的组合，这个数达到了 O（n^2）数量级。同样，在有 n 个程序员的团队中，相互联系的方式也是 O（n^2）数量级。而他们可支配的总共时间是 O（n）数量级。这样随着团队规模的扩大，成员们不可避免地要把所有时间都花在互相交流上。在实际生活中，人有不同的防御过度交流的机制。其中的一种方式就是忽略收到的信息；另一种方式就是在没有别人打扰的时间（晚上或者周末）工作。

作为一个程序员，你经常会淹没于来自四面八方的信息。他们会通过电子邮件传播消息、往你的信箱里塞印刷资料、邀请你参加会议、给你打电话，更糟糕的是，他们可能会直接到

你的办公室或家里找你谈话。

如果程序员只是编写和测试代码（不久以前还是这样），那么市场上就会充斥超低价的新操作系统、应用软件、工具软件、游戏、教育软件等。实际上，几乎所有公众域软件和共享软件都是由那些没有受过多程序员间沟通问题拖累的人员所开发。

图 21-1 所示为一个很简单的模拟结果。假设一个程序员每天花 10 分钟和团队中的其他程序员交流，剩下的时间用于真正的编程。那么，编程时间乘以团队中程序员的数目就是这个团队的生产效率——团队每天的有效工作量。注意，根据这一假设，25 人的团队生产效率最高，如果人数再增加的话，生产效率就开始下降。

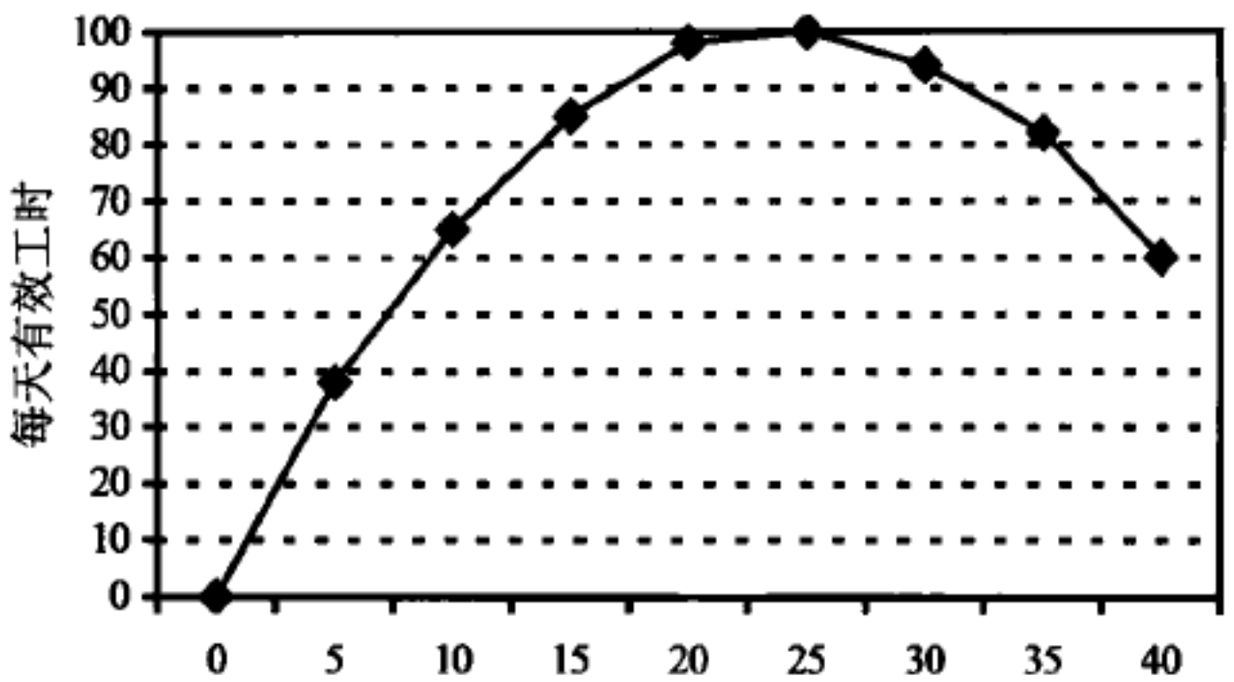


图 21-1 团队生产效率与团队规模的函数关系模拟

但还不仅仅如此!团队中的人越多，依赖关系图就越复杂。例如，组件 A 只有在组件 B 正常工作之后才能进行测试。组件 B 需要组件 C 的一些新功能，但组件 C 却在等待组件 D 排除错误。程序员总是在等，等久了就会失去耐性，于是发出更多的电子邮件，串门的人也多了。

还不能足以说明问题吗？再考虑一个有 n 个组件的系统的可靠性吧。组件越多，出错的可能性就越大。当一个组件出现故障，其他组件的测试要么不能进行，要么至少会受到影响。反过来，就会导致项目中出现更多的错误代码、甚至大的漏洞。这种互相影响的机制似乎还会进入一种恶性循环。

综上所述，图 21-1 所示的生产效率曲线显然太理想化了。有人认为，我们祖先的集体围猎比现代化工业更能决定团队的最佳大小，我看这也不无道理。

另一方面，如果你能通过提供更好的工具、更优秀的编程语言、更快速的计算机或者更满意的后勤保障来提供程序员的工作效率，你就可以建立一个正效应的反馈循环，使得团队的生产效率非线形提高。如果我们能在一个固定规模的项目中提高每个程序员的生产效率，那么团队就可以精简——进而减少通信开销，进一步提高每个程序员的有效生产效率。

每花一块钱用于提高单个程序员的生产效率，就可以节省雇用其他程序员所花的几块钱。

下面接着进行模拟，假如目标是在 500 天内写出 100 000 行代码。假设如果没有通信开销，每个程序员的生产效率是每天写 16 行代码。图 21-2 所示为随着生产效率的提高，团队规模相应缩小的曲线。

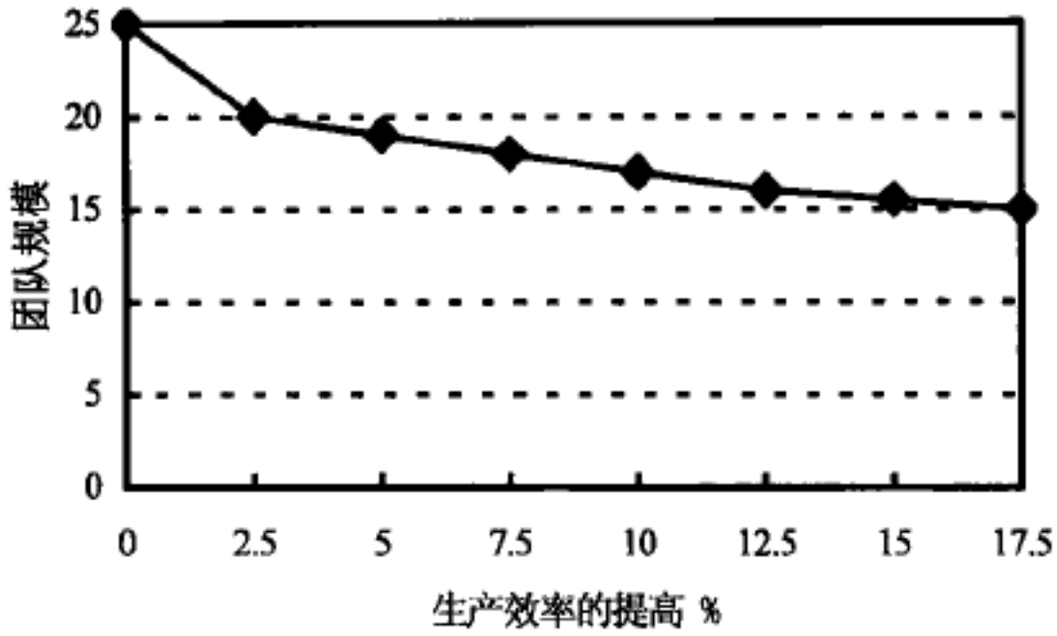


图 21-2　完成给定任务的团队规模与程序员生产效率函数关系的模拟

注意这条曲线趋近线性（大概对应 15 个程序员的情况）。从图中我们可以看出，单个程序员生产效率每增加 3%，就可以节省一个程序员。节省下来的程序员可以用于其他项目的开发。

影响生产效率的几个因素：

- 编程语言和方法的选择。到目前为止，在一般的编程任务中，C++和面向对象设计方法几乎是不二的最佳选择。如果不考虑速度和大小因素的话，也可以用选择其他一些专业化高级语言（如 Smalltalk、Prolog、BASIC 等）。所花费的代价就是程序员的初始培训费用。

- 项目编程约定的选择。例如是否使用异常、如何传播错误、代码共享策略是什么、如何处理整个项目范围的头文件等，如果你选择的不合适，就会对开发造成不利影响，并且在开发过程中很难再改正。所以最好在开始之前就深思熟虑。

- 编程环境和工具的选择。

- 硬件的选择。内存和硬盘容量至关重要。一个能收发电子邮件和共享打印机的局域网必不可少。还有，不要低估大屏幕显示器的重要性。

21.2　团队策略

在理想情况下，我们把整个工作分配若干个小组，每个小组均提供一个清晰、不变的接口供其他小组编写相应的代码。为每个接口定义一个全面的、详细的契约。小组之间的交流

减少到只有接口规范的交换，以及接口契约的定期更新。

在某种程度上来说，当开发团队使用外部创建的组件，如库、操作系统的 API（应用程序接口）等时，这种理想情况才会出现。任何接口和契约容易清楚描述的对象均可考虑放到库中。例如，字符串操纵库、容器对象库、iostreams 等在联机帮助、编译器手册以及各种书籍中都描述得很清楚。有些 API 函数不容易清楚描述，所以使用它们时需要反复地尝试和排错（或者客户支持调用）。

实际情况要比这复杂得多。是的，我们把工作分配给不同的小组，他们也都很努力地编写一些接口和契约。但是，这些接口不可能永远不变，契约也不可能完美无缺，它们需要小组间的反复讨论才能确定。我们所能做的就是使这个过程尽可能的顺利。

首先要有一个好的设计。花尽可能多的时间设计一个好的产品层次结构。然后，按照这个结构把小组分成不同的层次。结构设计得越好，产品各部分之间的接口和契约就越清楚。这意味着在后面的实现阶段改动和讨论越少。

小组的工作分配与设计结构明确对应，同时考虑相互间的通信。一旦设计结构明确了，立即把所有顶层组件的开发任务分配给各小组长。让他们协商组件间的契约。然后每个组长在他自己的小组内重复这个过程，设计组件的结构，如果有必要的话，再把子组件分配给若干子小组组长。

整个设计要经历几个阶段。底层设计的结果反馈给顶层组件的设计，最终体现在整个产品的设计中。每个小组都要写自己的规范部分，负责同一层次其他组件的小组要审阅这些规范。

在设计阶段的预先协商越多，实现阶段就越顺利。每次参与协商的人数要限制在一定的数量内。在全体会议上要向每个成员描述产品的顶层设计必不可少，让每个人对全局都有一个大致的了解。在实现阶段也可以召开全体会议，监督产品的开发进度。不过要注意，全体会议不适宜用来讨论具体的设计。

在实现阶段的契约协商可能会是这样：A 组的一些成员根据他们对契约的理解使用 B 组的一个接口。接口代码意外出现错误，如碰到断言、抛出异常或引起混乱。那么 A 组成员去找 B 组的组长，询问是谁负责实现此接口，或者直接去找接口的实现程序员，探明发生奇怪行为的根源。实现者或者进一步明确契约，根据 A 组的需要对接口修改实现代码完善契约，或者代 A 组成员查找漏洞。如果修改 B 组件，那么就要对它进行全面的测试，看看是否会影响其他使用此组件的用户。

在实现一些主要新功能时，可能需要请求其他组修改或扩展他们的接口和（或）契约。这可以看作是重新设计。和系统中的其他干扰一样，重新设计也会在系统中引起连锁反应。必须重复在前面初始设计阶段所描述的反馈过程。接口和契约的改动影响首先在那些直接与之关联的小组中传播（因而，他们要确保改动是必须的，并要详细描述这些改动），然后将改动通知顶层设计。不过在向顶层传播的过程中，有些组件可以通过内部修改消化这些改动，

这样顶层设计就不用修改了。否则，这种变动可能会传播到最顶层，进而影响顶层对象之间的交互方式。这种项层对象的改动又会向下传播到所有相关小组。下层修改再反馈给顶层，这一过程多次往复，直到改动的组件都稳定为止。也就是说，当项目达到一个新的平衡状态时，这一“退火”过程才会结束。

第22章 实现策略

一旦完成设计任务，接下来的程序开发应从何处着手呢？项目的各个部分该依照什么顺序来实现呢？处理修改的最好方法是什么？虽然这些问题对不同的项目有不同的答案，但在项目的准备和实现过程中，还是有一些一般性规则可以遵循的。

22.1 确定全局

错误处理、异常、普通头、代码重用、调试输出。

在实现之前要做的最大的全局决定就是如何处理错误和异常。主要错误来源有：

- 有错误的组件
- 其他（受信任）组件传递过来的错误参数
- 不正确的用户输入
- 永久数据结构崩溃
- 系统资源耗尽

程序的最终零售版本中不应该存在错误，所以我们应该在开发过程将它们都消灭掉（当然，市场上大部分零售软件还是有一些残留的错误）。因为在开发过程中，我们经常主要应用调试构建机制，所以，我们可以在代码中插入断言语句来避免错误。断言还可以强制执行组件之间的契约。

对于用户输入（一般针对来自不太可靠系统的输入），在进一步应用之前，必须全面测试其正确性。为了确保程序不被致命输入击跨，必须进行随机性的输入测试。如果我们的程序为其他程序提供服务，则必须测试外部传递参数的有效性。例如，操作系统 API 函数一般都会对应用程序传递过来的参数进行检查。同时还要进行参数类型错误检查。如果它是用户直接输入的，我们应该立即作出反应；如果它是从一个不可靠的组件传递过来的，我们应该返

回相应的错误代码或者产生相应的异常。

任何没有完全置于我们控制之下的永久数据结构经常会遭到其他应用程序和工具的破坏（实际情况也经常是这样，除非我们本身就是文件系统），更不用说硬件故障和用户错误了。所以，我们总是应该测试它们的兼容性。使用断言保证从磁盘读取数据结构的一致性是一个常规编程错误。磁盘上的数据从来都不可靠，所以在零售版本中必须进行彻底的检查。

不要用断言来保证永久数据结构的一致性。

系统资源——内存、磁盘空间、句柄等的耗尽是产生异常的主要原因。以内存为例。假如要求所有的程序员检查非抛出运算符 new 的返回值（这是不现实的）。那么，如果返回指针为空该怎么办？这就需要具体情况具体分析。每一次调用 new，程序员都应该提供一些合理的恢复措施。然而，很少测试恢复路径（除非测试小组能模拟所有类型的失败）。所以，程序员应该花一些时间来设计模拟发生故障的程序。

处理内存耗尽的最简单方法是打印一条“内存耗尽”消息并退出。通过设置自己的内存耗尽处理器（C++中的 set_new_handler 函数），这一点很容易完成，不过很少有人这么做。在大部分内存耗尽的情况下，至少要做一些清理工作，把用户数据保存到磁盘上，甚至返回到程序的上一层尝试继续运行。看来最恰当的处理方法是使用异常和资源管理技术（参见本书前面内容）。

如果没有 C++异常处理或者经理禁止使用它，我们只好使用传统技术了，测试 new、清理以及错误向上传递的结果。当然，必须使用失败模拟全面测试程序。正是这种做法形成了在整个项目中“每个函数必须返回一个状态码”的约定。普通返回值由引用或者指针传递。状态码系统很快发展成一个拜占庭式的错综复杂结构。实际上，在这种方案中，每个错误代码不仅指向错误处，而且还包含错误的全部历史，因为在错误传递的每个阶段都同时包含了对错误的解释。构造函数的使用受到严格限制，因为它是唯一不能返回值的函数。缺乏处理复杂情况的能力，C++很快就退化成“改良的 C”。幸运的是，大部分现代 C++编译器都支持异常处理，很快我们在这里的讨论将成为历史。

另一个要提前做的重要决定是，选择整个项目范围内的调试约定。通过调试输出或者日志很容易知道其进度和状态消息。

接下来是目录结构和构建过程。项目及其组件结构必须反映在源代码目录结构中。还需要安排放置全项目头文件和代码的位置。在这里，程序员还要放置调试工具、公共类型定义、全项目范围的参数、共享应用代码以及有用的模板等。

在项目中有必要进行一定程度的代码重用，而且应该精心组织。对可重用代码及其文档的宣传通常容易被忽视。应该定期通告可重用部分的消息，时刻提供最新说明文档。

还有一点：在 C++中，头文件和实现文件之间紧密相连——不查看和修改其中一个，就很难成功修改另外一个。这也是在大部分情况下把这两者放在同一个目录下，而不用单独包

含目录存放的原因。不过，不同目录之间共享头文件例外。

把与平台有关的层单独放在不同的目录中是个好主意。我们将在第 23 章“移植”中讲述这方面的内容。

22.2 自顶向下的面向对象实现方法

实现过程应尽可能模仿设计过程，这是为什么要从顶层组件开始实现的原因。对顶层接口的修改越早就越好。另外，我们需要尽快有一个测试程序。

原始实现过程的目标是测试控制流程、生存期和顶层对象的可访问性以及初始化和关闭过程。在这个阶段，不要编写任何实质性的代码，不能用于演示，也不能作为原型。

> 如果经理要一个程序原型，那么应该由一个单独的小组实现，可以用不同的语言（BASIC、Smalltalk 等等）来写。在主项目中试图重用为原型编写的代码是一个大错误。

在这个时间只要实现那些必不可少的基本功能就行了，其他功能以后再完善。此时的类方法必要时只打印调试信息和显示参数。调试和错误处理装置应该到位并经过测试。

如果程序是交互性的，我们应该实现必需的视图和控制器部分来获得流向模型的信息并显示尽可能少的视图。可以完全忽略模型部分。

一旦确定了程序的工作骨架，我们就可以着手实现低层对象。在每个阶段重复以上基本步骤。首先是创建该层所有对象的部分代码，然后测试它们的接口和交互作用。一直重复这个过程直到项目最底层。在最底层，我们开始实现“真正”的功能。最底层组件的目标是能够符合整个结构。这些组件应当各就各位、添加适当的控制代码、设置适当的调用参数，而且返回正确的值。

采用这样的实现策略，得到的将是具有同样水准的程序，组件之间配合流畅、天衣无缝，就像一辆精心设计的跑车一样。相反，采用自底向上实现的程序各部分质量参差不齐、形如散沙，就像用胶带和绳子强行绑在一起一样。大部分程序员的时间花费在“把方木桩插到圆洞中去”。得到的程序什么都像，就是不像“一辆精心设计的跑车”，程序的质量可想而知！

22.3 程序修改

代码修改的范围从简单的修饰性修改（如重新命名变量）到彻底性改动以及重写主要部分。通常在代码审查期间作一些小量修改。根据经验，如果一个函数有太多的本地变量或者对象，或者相互间传递的参数太多，那么这部分代码应该重新进行抽象。

有趣的是，最顶层和最低层会变得与面向对象模式不太符合。要设计出一套恰到好处的顶层对象通常比较困难，最后得到的主函数也通常相当庞大。相反，在层次的底端，却没有

使用大量短生存期的轻型本地对象的优良传统。顶层的状况由设计的好坏决定，而底层的情况很大程度上依赖代码审查的质量。上面说的经验规则在这里很有帮助。同时要随时注意不要使用太多的“剪切—粘贴”式重复代码。如果差别细微的同一组操作在很多地方都出现的话，就应该采用一个新的抽象了。

重写小部分程序是良性开发的标志。有时重写的意义更重大。它们可能关系到抽象某个层，此时，必须修改指定服务的所有客户程序，或者改变程序的高层结构，此时，大量低层结构都受到影响。幸运的是，在自顶向下面向对象设计中，这样彻底的改动较易实现。可以将一个顶层对象分解成若干个更加独立的部分，或者更改最高层的容器和访问结构（例如把一个子对象从一个对象移到另一个对象中）。如何实现呢？关键是自顶向下一步一步地修改。

第一遍，修改接口并传递不同的参数集，例如，把引用变量传递到原来可以直接访问某些对象，而现在却禁止访问的地方。尽可能少地修改实现代码。最后，编译并测试。

第二遍，考察每个对象，看它是否能够访问所需要的数据。编译并测试。

第三遍，一旦所有对象以及参数准备就绪，就开始一步一步对实现代码进行必要的修改。

从一开始编写实现代码就要同时进行测试。不管什么时候，你都要有一个工作程序，不光你可以用它来进行测试，你的同事也需要它来进行测试。也许这个工作程序开始时尚没有达到预料功能性，但它能运行而且至少能打印一些调试信息。一旦开始实现某些功能，就要开始回归测试。

在回归测试中，开发一个测试套件测试系统的基本功能。在每一次改动之后，都运行这个套件看看程序功能是否遭到破坏。每增加一个新函数，都要相应扩充测试套件。回归测试不会耗费太多时间。

当某些程序功能接近最终目标时，立即开始进行重点测试。和回归测试不一样，重点测试是用来测试系统的极限性。例如，全面测试可能失败的场合、不同处的内存耗尽错误、磁盘错误、意想不到的断电等。

还应该测试程序在重负荷工作状态下的伸缩性。根据程序的不同类型，重点测试可以通过处理许多小文件、一个特大文件、许多请求等来完成。

22.4　代码继承

（程序员眼中的）理想世界是，所有项目都是从头开始，没有任何外部依赖性。这种情况偶而会发生，此时，技术发出了很大的变化，可以应用和测试新语言、新编程方法以及新的团队结构。

而在现实中，大多数项目都要继承某些源代码，这些源代码通常使用了过时的编程技巧，它们有自己的错误处理、调试模型，应用（或误用了）全局对象、goto 语句、杂乱的代码、

代码篇幅成页成页的函数等。大部分程序都存在外部依赖性——不同开发商开发的代码、工具或库。最糟的是，这些开发商的开发目的不一样，他们要发行自己的产品，在市场上竞争。在大多数情况下，他们都会很热情地欢迎你使用他们的代码或工具，并保证长期提供技术支持。但不幸的是，他们有自己的优先服务对象。所以要确保你们的管理部门和他们的管理部门之间沟通良好。

如果你能全面控制所继承的代码，则可以计划自己一步步重写它。一遍遍审查代码，找出最容易出问题的地方，首先重写这部分代码。然后进行平行开发，交织进行新代码开发的重写。你可能觉得这样做太浪费时间，其实不用担心，这里所花的时间换来的是调试时间的减少和代码整体质量的提高。

22.5 多平台开发

有很多应用程序同时为多个平台（如不同的硬件和不同的 API 函数集）一次性开发。操作系统和计算机在不断发展——在任何时候都有过时的平台、流行的平台以及下一代平台。有时，目标平台和开发平台并不相同。在任何程序中，与平台有关的部分都要提取出来，放入单独的层或者库中。

操作系统应该提供一个抽象层，把应用程序和硬件隔开。除了非常专业化的应用程序外，磁盘访问很适合提取出来放入文件系统中。在窗口系统中，把图形和用户输入提取出来放入窗口 API 函数中。我们的程序也应该这样，把所有与平台相关的部分都提取出来，放入一个单独的层中。这个层是一套服务的集合，通过它们我们可以访问底层功能。分层的好处就是，修改层时，不必修改使用它的代码。而且，我们还可以添加新的实现代码，或者使用编译时变量从一个平台转换到另外一个平台。有时，一个平台没有提供、甚至还需要其他平台提供的功能。例如，一个非多任务操作系统就不需要信号量，但我们仍然可以提供一套加锁系统，根据不同的平台把它打开或者关闭。

我们可以通过创建一套抽象出某些功能的类来构建层。例如，内存映射文件可以和缓冲文件在一个抽象层中结合使用。选择这样的实现方法是明智的，也最能适应未来平台的要求。

还有一点值得注意，如果不同平台上中基本数据类型大小不一样，例如，有的平台为 16 位整数，有的是 32 位。我们在设计永久数据结构以及可以通过网络传输的数据类型时要特别小心。万无一失的防范措施是把所有的基本数据类型都转化成字符串类型，字符串的字节数统一定义。

通过这个方法，我们甚至可以化解 big-endian 和 little-endian 之间的差异。不过，由于这种方案会增加运行时开销，因此也并不总是可行。折衷的办法是：要么只支持那些短型数与长型数大小兼容（末端也兼容）的平台，要么提供把永久数据从一种格式翻译成另外一种格式的转换程序。

第23章 移植

在一个软件工程师的职业生涯中，从头编写大型代码的机会并不多。所以一旦有这样的机会，就应该尽情享受、尽情发挥。通常情况下，程序员的工作是维护、重写现有代码或者把程序移植到新平台。程序移植可能是让程序运行于不同的硬件、不同的操作系统，或不同的 API 函数集，不同的移植目的复杂程度也不同。实际上，程序移植是一项很普遍的工作，所以我决定用一个程序移植的例子来结束本部分内容。

23.1 创建一个抽象层



Project PortCalc

移植是重新构建代码的绝佳机会。在进行程序移植时，你的首要目标是分离出与平台有关的实现部分，并把它们隐藏在一个抽象层中。程序的大部分只需要调用这个隐藏层而不需要知道它的实现细节。把程序移植到新平台要重写抽象层本身，而不必改动其余代码。

抽象层通常通过一套访问子系统的接口来定义。它作为客户和底层实现代码之间的一道防火墙。抽象层的客户只能通过这些公共接口访问子系统。只要接口保持完整以及两边都遵守契约，防火墙两边的代码可以自由发挥。

成功移植后，系统可以继续以平台无关方式开发。如果要添加与平台相关的功能，那么应该在移植层中实现。在大多数情况下，移植层还有一个额外的功能，它可以使代码的可读性和可维护性更好。

和往常一样要避免不成熟的优化。到处放置 ifdef（编译指令）似乎比应用抽象层更好。除非你有无可辩驳的证据表明，有问题的代码就是程序的主要瓶颈，此时才迫不得已使用这一方法。不然，维护程序时，你和你的同事将付出艰辛的劳动。

23.2 移植计算器

当我们把计算器的命令行版本转化成 GUI 界面时，我们其实已经进行了某种移植。不过，我们进行的是单向移植——重写之后的程序不能再被编译成命令行形式了。在本章，我们将做一些更一般性的移植工作。

我不想在本书中再引入另外一套 API 函数，所以，我不想移植到任何特定系统。完成第一阶段移植（将与 Windows 相关的功能分离到一个单独层）就足够了。针对 Linux 或者 Mac API 函数重写这个层是另外一个问题，完成这部分移植需要对这些系统有一定的了解，有兴趣的读者可以自己练习。

到目前为止，我们已经可以很好地封装 Windows API 函数。Windows 库的接口也形成了一个合理的抽象层，所以我们现在的任务相当轻松。首先，任何非库文件不要包含<windows.h>，或者直接调用任何 Windows API 函数。我们这样做的目的其实也不完全是为了移植方便，主要原因是老式的低层 Windows API 函数与现代编程方式不兼容。

接下来，我们要确保<windows.h>中定义的任何数据类型没有直接应用于源代码中。检查工作要逐步完成。接着，为了证明其余代码部分完全与 Windows 无关，我们将创建一个不包含任何 Windows 头文件的简短库，并且使用这个库编译程序。根据不同的平台，在必要时还要继续充实这个库。

23.2.1 消除简单的 Windows 依赖性



Script 1

在第一次大规模代码重写中，我们把注意力集中在 UINT、WPARAM、LPARAM、不同句柄、位标志等这样一些 Windows 特有的数据类型上。

除了用户自定义消息外，大部分 Windows 消息处理过程完全封装在一般性窗口过程中。不仅 Win::Controller::OnUserMessage 方法直接把原始 Windows 数据类型作为参数，而且用户也被迫定义此类特定消息，如根据 WM_USER 这样的 Windows 消息来定义 UM_MEMCLEAR 消息。

我们可以创建一个 Win::UserMessage 库类来进行弥补，此库类知道如何处理消息编号、参数打包的底层细节。尤其是，这一库类可以接受以 0 为基数的消息编号 ID，并且在内部把 Windows 特定的常量 WM_USER 加到 ID 上。用户消息的发送通过 Win::Dow::SendMsg 的重载来完成，这一重载带有 Win::UserMessage 参数，并进行必需的类型转换。

```
void Win::Dow::SendMsg(Win::UserMessage & msg) const
{
  LRESULT result = ::SendMessage(_h,
                                 msg.GetMsg() + WM_USER,
```

```
                                    msg.GetWParam(),
                                    msg.GetLParam());
  msg.SetResult(result);
}
```

因此，Win::Controller 的 OnUserMessage 方法现在也可以根据 Win::UserMessage 来定义。

还有一个地方需要显式发送 Windows 消息——当我们对编辑控件进行子类操作截取回车键并把它转换成一个命令消息时。同样，我们可以隐藏把参数打包到库类 Win::CommandMessage 中命令消息的机制。在这里，很明显应该创建一个更一般的类 Win::Message。它将作为 Win::UserMessage、Win::ControlMessage 或者在代码中需要显式使用的其他任何消息的父类。



Script 2

我们的菜单子系统有几处明显的 Windows 依赖性。例如，OnInitPopup 方法使用 HMENU 作为参数。用 Menu::Manip 类代替 HMENU 就可以解决。



Script 3

TopController::OnMenuSelect 方法使用了整套 Windows 特定常量，比如 MF_SEPARATOR、MF_POPUP 等。我们不仅可以把这些常量隐藏在位标志 Menu::State 中，也可以用两个单独的 Win::Controller 方法来对应菜单被选中和菜单未选中这两种情况。记住：移植可以使程序得到更好的抽象。



Script 4

PostQuitMessage API 函数可以封装到一个自由函数 Win::Quit 中。



Script 5–9

OnSize 处理器也用了一些 Windows 特有的标志。我们将把它们封装到一个枚举类型 Win::SizeType 中。同样，我们还将枚举列表框通知代码、虚拟键代码以及几个不同种类的 Windows 常量。



Script 10–14

如何处理 HINSTANCE 现在还不完全清楚。它是一个 Windows 特有的数据类型——一个程序实例的句柄。不幸的是，“程序实例”似乎并不代表一个具有一般性的抽象，要把它移植到其他系统上也不容易。它经常被用来确定装载程序资源（图标、位图等）的位置。在我们有限的用途中，它通常指当前运行的可执行程序、在 Winmain 中传递给我们的句柄以及其他可用作资源的对象。因为没有更好的办法，所以我用一个更中性的类型 Win::Instance 来代替 HINSTANCE。在我们的 Windows 库中，它只是 HINSTANCE 的一个类型定义。在移植过程

中，可以把它转换为任何适当的类型。

显然，WinMain 作为 Windows 特有的入口必须保留在我们的代码中。程序的移植版本必须有它自己的入口。为了使移植更简单，我还创建了一个与平台无关的入口，这就是 Main，它可以从其他任何平台专有入口（尤其从 WinMain）调用。



Script 15

消除简单 Windows 依赖性的最后一步是创建简短库。这一步比较简单，首先把所有 Windows 特有的包含文件（尤其是<windows.h>）从库中删掉，并删除库中所有方法的实现代码。经过这一步之后，如果程序仍然能编译（虽然不能运行），那么我们就成功地把 Windows 相关的部分去掉了。

相关测试可以保证我们在程序中使用的库接口确实是与 Windows 无关的。这个接口现在可以使用不同的 API 集来实现——充实简短库，但要保证新旧实现版本的功能一样，这样我们的程序就可以在新环境中运行了。

23.2.2 特殊 Windows 依赖性

现在我们已经把程序中比较明显的 Windows 依赖性都消除了，下面让我们来看看那些不是很明显的依赖性。

首先要说明，不能保证不同的窗口 API 集一定可以抽象为适合于 Windows 的类和方法。幸运的是，从 20 世纪 70 年代末起，所有基于窗口的商业 GUI 系统都有一个共同的起源，Xerox PARC（Palo Alto Research Center 之意）Smalltalk 系统，即 Alto。这样我们就可以放心地假设我们遇到的任何窗口 API 集都具有窗口、窗口类、消息处理器、菜单、对话框、鼠标指针等的抽象。很可能你还可以找到许多标准 Windows 控件，如按钮、编辑控件、列表框等的实现。至于那些没有的控件，你可以用已有的基本控件来实现。

Windows 中对话框的定义方式使得它不太便于移植。不仅资源脚本使用的是专有语言，而且对话框中各个元素的放置方式也是不可移植的。Windows 资源脚本指定了各种控件的大小和位置。不过，所用的单位不是像素而是特殊的对话框单位。对话框单位是以它所用的字体的平均大小为基础的。

这种做法可以使对话框的布局与屏幕分辨率以及字体大小无关。当 Windows 显示一个对话框时，它根据对话框所用字体的尺寸把对话框单位转化为像素。这样，文本框以及其他控件可以根据字体大小按比例缩放。不过在实际应用中效果不是太好，甚至在微软的不同操作系统中效果也不理想。在 Windows98 下设计的对话框在 Windows NT 中显示时，文本框总残缺不全，反之亦然。

设计对话框的更好途径是用一种更抽象的方式来指定控件的布局。这方面可以借鉴 HTML 和 Java 的经验。例如，在 Java AWT（Abstract Window Toolkit，抽象窗口工具包）中，

格式化是通过不同的 Layout 对象动态完成的。

通过对话框布局的动态实现，我们可以使库的移植性更好、更容易使用。确实，在 Windows 程序中，可以动态创建对话框模板，然后调用 API DialogBoxIndirect 显示它。这样就可以绕过不可移植的资源脚本，并且在设计和修改对话框时具有更多的灵活性。

23.3　移植的好处

在完成计算器移植之后，我更喜欢这一程序了。在库和应用代码之间有明显的差别。即使无需移植此计算器程序，我也要保留修改后的库接口。注意，每次我们一般化程序时，都对它的结构做了改进。在整本书里，我不停地对这个计算器程序进行修改，每次修改之后它的代码看起来都更有逻辑性了。在许多情况下，一般化可带来简洁（例如使用动态数据结构以及标准库）。

这也许和一般的经验背道而驰。一般人们都认为修改会增加复杂性并且破坏程序的结构，而不是与之相反。大多数程序随着开发的深入，可维护性变得越来越差。至于软件项目的混乱度极限也无法根据什么定律来确定。复杂性可通过强行调整结构来消除。如果每一次改动都伴随着相应的结构改动，那么软件项目可以永远保持进化。

很多开发团队常犯的一个错误是在结构修改方面不够积极。给程序加更多的特例、传递额外的标志，调用某过程完成一些额外工作而不是重新构思一个抽象、添加新层，或更合理化某过程的实现逻辑，似乎前者能快速达到目的，后者不能立竿见影。所以一般在期限比较紧、压力比较大的时候，很难在已经很紧的日程表上挤出一点时间来进行这样的修改。不过从长远来说，这种工作是必需完成的。

本书的大部分内容都是关于程序修改的，这绝不是偶然。如果在修改程序时，能够既不牺牲结构，又能确实提高性能，这也许就是一个程序员相对于开发团队最重要的技巧了。

附　录

附录A

练习答案

附录A提供本书第一部分内容中部分练习的答案。有些答案，我描述比较详尽，而有些答案只提供了提示。从随书光盘上可找到部分答案源代码。我还将在网站www.ReliSoft.com/book上发布更多的答案。

第1章

1．第一种方案是在一个循环中创建两个world：

```
for(int i = 1; i < 3; ++i)
  World world(i);
```

第二种可能方案是对每个world开辟一个新的单独本地作用域：

```
int main()
{
  {
    World world(1);
  }
  {
    World world(2);
  }
}
```

2．错误在于数据成员初始化的顺序。尽管在构造函数前导中，初始化器按序排列，但根据它们在Glove中的嵌入顺序，_hand仍然在_n之前初始化。因此，_n不应当作为参数传递给 Hand 的构造函数，因为此时还没有设置它的值。更正的一种办法是改变嵌入的顺序。另一种办法是使用numFingers代替_n来初始化_hand。

3．类Frame

```
class HorBar
```

```
{
public:
  HorBar(int n)
  {
    std::cout << "+";
    for(int i = 0; i < n; ++i)
      std::cout << "-";
    std::cout << "+\n";
  }
};

class VerBar
{
public:
  VerBar(int n)
  {
    for(int i = 0; i < n; ++i)
      std::cout << "|\n";
  }
};
```

```
class Frame
{
public:
  Frame(int hor, int ver)
  : _upper(hor),
    _ver(ver),
    _lower(hor)
  {}
private:
  HorBar _upper;
  VerBar _ver;
  HorBar _lower;
};
```

4. 类 Ladder

```
class Ladder
{
public:
  Ladder(int hor, int ver)
  : _upper(hor, ver),
    _middle(ver),
    _lower(hor, ver)
```

```
  {}
private:
  Frame    _upper;
  VerBar   _middle;
  Frame    _lower;
};
```

5. 本答案只使用第 1 章介绍的结构。

```
int main()
{
  InputNum num;
  int factorial = 1;
  for(int i = 2; i < num.GetValue(); ++i)
    factorial = factorial * i;
  factorial = factorial * num.GetValue();
  std::cout << num.GetValue()<< "! = "
            << factorial << std::endl;
}
```

6. 类 Planet

```
class Planet: public CelestialBody
{
public:
  Planet(double mass, double albedo)
    : CelestialBody(mass),
      _albedo(albedo)
  {
    std::cout << "Creating a planet with albedo"
        << _albedo << std::endl;
  }
  ~Planet()
  {
    std::cout << "Destroying a planet with albedo"
        << _albedo << std::endl;
  }
private:
  double _albedo;
};
```

7. 第一部分：

```
class Two
{
public:
  Two()
  {
```

```
    std::cout << "Program";
  }
};
class Three
{
public:
  Three()
  {
    std::cout << "objects ";
  }
};

class Four
{
public:
  Four()
  {
    std::cout << "makes ";
  }
};

class One: public Two
{
public:
  One()
  {
    Three three;
    std::cout << "with class. ";
  }
private:
  Four    _four;
};
```

第二部分可以实现如下：

```
class One: public Two
{
public:
  One()
  {
    std::cout << "class ";
  }
  ~One()
  {
    std::cout << "class ";
  }
```

```
private:
  Three  _three;
  Four   _four;
  Five   _five;
};

int main()
{
  One one;
  std::cout << std::endl;
}
```

其中类 Two、Three、Four 和 Five 分别打印“Program”、“makes”、“objects”及“with”。

8. 类 Average

```
class Average
{
public:
  Average()
    : _sum(0), _count(0)
  {}
  void Put(double n)
  {
    _sum = _sum + n;
    ++_count;
  }
  double Get()const
  {
    return _sum / _count;
  }
private:
  double    _sum;
  int       _count;
};
```

9. 类的层次如下：

```
class LivingBeing {};
class Human: public LivingBeing {};
class Tomato: public LivingBeing {};
class Elephant: public LivingBeing {};
```

或者更彻底，将植物从动物中分离出来：

```
class LivingBeing {};
class Plant: public LivingBeing {};
class Animal: public LivingBeing {};
class Human: public Animal {};
class Elephant: public Animal {};
```

```
class Tomato: public Plant {};
```

10. 练习中所描述类的骨架如下：

```
class NoseHair {};

class Nose
{
...
private:
  NoseHair _noseHair;
};
class Head
{
...
private:
  Nose _nose;
};

class Human
{
...
private:
  Head _head;
};
class Man: public Human {};
class Woman: public Human {};
```

11. **Methods Top and Const**

```
int IStack::Top()const
{
  assert(_top > 0);
  return _arr [_top - 1];
}

int IStack::Count()const
{
  return _top;
}
```

12. 此例对空堆栈调用 Pop，违反了契约：

```
int main()
{
  IStack stack;
  stack.Pop();
}
```

13. 以下是字符堆栈的接口：

```
class CharStack
```

```
{
public:
  CharStack(): _top(0){}
  void Push(char c);
  char Pop();
  char Top()const;
  int Count()const;
private:
  char _arr [maxStack];
  int  _top;
};
```

使用字符堆栈颠倒字符串：

```
int main()
{
  CharStack stack;
  char str [] = "esreveR";
  for(int i = 0; str [i] != '\0'; ++i)
    stack.Push(str [i]);
  while(stack.Count()> 0)
    std::cout << stack.Pop();
  std::cout << std::endl;
}
```

14. 类 Queue

```
#include <iostream>
#include <cassert>

const int maxPuts = 8;

class Queue
{
public:
  Queue();
  double Get();
  void Put(double x);
private:
  double _arr [maxPuts];
  int    _putIdx;
  int    _getIdx;
};
Queue::Queue()
: _putIdx(0),
  _getIdx(0)
{}

double Queue::Get()
{
```

```
  assert(_getIdx < _putIdx);
  ++_getIdx;
  return _arr [_getIdx - 1];
}

void Queue::Put(double x)
{
  assert(_putIdx < maxPuts);
  _arr [_putIdx] = x;
  ++_putIdx;
}
int main()

{
  Queue queue;
  queue.Put(0.1);
  queue.Put(0.2);
  std::cout << "Getting: " << queue.Get()<< ", "
          << queue.Get()<< std::endl;
  queue.Put(0.3);
  std::cout << "Getting more: " << queue.Get()<< std::endl;
}
```

15. 类 DblArray

```
#include <iostream>
#include <cassert>

const int maxCells = 16;

class DblArray
{
public:
  DblArray();
  void Set(int i, double val);
  double Get(int i)const;
  bool IsSet(int i)const;
private:
  double _arr [maxCells];
  bool   _isSet [maxCells];
};

DblArray::DblArray()
{
  for(int i = 0; i < maxCells; ++i)
    _isSet [i] = false;
}
```

```
void DblArray::Set(int i, double val)
{
  assert(i < maxCells);
  assert(!IsSet(i));
  _arr [i] = val;
  _isSet [i] = true;
}
double DblArray::Get(int i)const
{
  assert(i < maxCells);
  assert(_isSet [i]);
  return _arr [i];
}

bool DblArray::IsSet(int i)const
{
  assert(i < maxCells);
  return _isSet [i];
}
int main()
{
  DblArray arr;
  arr.Set(2, 0.2);
  arr.Set(4, 0.4);
  std::cout << "IsSet(0)returns " << arr.IsSet(0)<< std::endl;
  std::cout << "IsSet(2)returns " << arr.IsSet(2)<< std::endl;
  std::cout << "Get(2)returns " << arr.Get(2)<< std::endl;
  std::cout << "Get(4)returns " << arr.Get(4)<< std::endl;
}
```

第2章

1. 需要注意的运算符有：

- 一是补码运算符。它是一个一元运算符，所以，你应该只从堆栈中弹出一个参数。
- 其次是左移位和右移位运算符。这是双字符运算符，所以，发现小于号或大于号后，确保输入缓冲器中的下一个字符与第一个字符相同。你可以把第一个字符作为标记。

2. 给 Calculator 添加布尔数据成员：

```
bool _isHex;
```

（记住在构造函数中初始化）在 Execute 方法中，检查标记'x'并切换标志：

```
else if(token == 'x')
{
  // Toggle hex display
  _isHex = !_isHex;
```

```
    status = true;
  }
```

在 main 中，显示堆栈内容前检查此标志：

```
if(TheCalculator.IsHex())
  std::cout << std::hex;
else
  std::cout << std::dec;
```

3. 不管在计算中哪儿保存或使用值，都必须将类型 int 转换成 bool。尤其是重写 IStack（叫做 BStack）来保存布尔值。Calculator::Execute 的代码片段如下：

```
bool b2 = _stack.Pop();
if(token == '!')
{
  _stack.Push(!b2);
  status = true;
}
else
{
  bool b1;
  //特例，仅当堆栈中只有一个数时
  //两个操作数均使用这一个数
  if(_stack.IsEmpty())
    b1 = b2;
  else
    b1 = _stack.Pop();

  _stack.Push(Calculate(b1, b2, token));
  status = true;
}
```

Calculate 方法如下：

```
bool Calculator::Calculate(bool b1, bool b2,
                           int token)const
{
  bool result;
  if(token == '&')
    result = b1 && b2;
  else if(token == '|')
    result = b1 || b2;
  return result;
}
```

第 3 章

1. 一种可能方案：

```
void StrCpy(char * strDest, char const * strSrc)
```

```
{
  int i = 0;
  char c;
  do
  {
    c = strSrc [i];
    strDest [i] = c;
    ++i;
  } while(c != '\0');
}
```

2. 请注意，此处同一个条件（str2 短于 str1）对应两个 case。一般，不推荐使用这种优化，因为它使代码变得晦涩。

```
int StrCmp(char const * str1, char const * str2)
{
  int i = 0;
  while(str1 [i] != '\0')
  {
    int diff = str1 [i] - str2 [i];
    // 也对应于 str2 的结尾
    if(diff != 0)
      return diff;
    ++i;
  }
  if(str2 [i] == '\0')
    return 0;
  // str1 更短
  return -1;
}
```

3. 下面是途径之一。不过，其中的定义不完整。我没有指定当源字符串的长度大于或等于 len 时该执行什么动作。目标字符串应该在偏移 len-1 处用 null 字符终止吗？你认为呢？

```
void StrNCpy(char * strDest, char const * strSrc,
             std::size_t len)
{
  for(std::size_t i = 0; i < len; ++i)
  {
    strDest [i] = strSrc [i];
    // early exit
    if(strSrc [i] == '\0')
      break;
  }
}
```

4.

```
#include <iostream>

int main(int count, char * argv [])
```

```
{
  for(int i = 0; i < count; ++i)
  {
    std::cout << i << ": " << argv [i] << std::endl;
  }
}
```

5. 不希望将堆栈压缩至_top，因为下一个推入操作必须再次扩展它。合理的方案是将堆栈的容量压缩至当前_top 的两倍。仅当_top 降低到当前容量的 1/3 时才进行压缩。

```
int IStack::Pop()
{
  assert(_top > 0);
  --_top;
  if(_top >= 1 && 3 * _top < _capacity)
    Shrink();
  return _arr [_top];
}

void IStack::Shrink()
{
  std::cout << "Shrinking stack from "
            << _capacity << " to " << 2 * _top << ".\n";
  assert(2 * _top < _capacity);
  // 分配新数组
  int * arrNew = new int [2 * _top];
  // 复制所有项
  for(int i = 0; i < _top+1; ++i
    arrNew [i] = _arr [i];
  _capacity = 2 * _top;
  // 释放旧内存
  delete []_arr;
  // 将旧数组替换成新数组
  _arr = arrNew;
}
```

6. 两级深度堆栈：

```
Stack2::Stack2()
  : _last(0), _previous(0)
{}

void Stack2::Push(char const * str)
{
  _previous = _last;
  _last = str;
}

void Stack2::Pop()
```

```
{
  _last = _previous;
  _previous = 0;
}

char const * Stack2::Top()const
{
  return _last;
}
```

7. 技巧是使用两个指针：一个指向列表头，一个指向列表尾。当删除元素时，使用头指针，添加新元素时使用尾指针。与往常一样，我们真正关心的是边界情况——队列起始为空和变成空队列。

```
void Queue::Put(int i)
{
  Link * newLink = new Link(i);
  if(_tail == 0)
  {
    _head = newLink;
  }
  else
  {
    _tail->SetNext(newLink);
  }
  _tail = newLink;
}

int Queue::Get()
{
  assert(_head != 0);
  int i = _head->Value();
  Link * tmp = _head;
  _head = _head->Next();
  // 不要递归删除!
  tmp->SetNext(0);
  delete tmp;
  if(_head == 0)
    _tail = 0;
  return i;
}
```

8. 解链接的列表序列生成器：

```
class UnlinkSeq
{
public:
  UnlinkSeq(List & list)
    : _list(list),
```

```
    _cur(list._pHead),
    _prev(0)
{}
bool AtEnd()const { return _cur == 0; }
void Advance()
{
  _prev = _cur;
  _cur = _cur->Next();
}
  int Id()const { return _cur->Id(); }
  void Unlink();
private:
  List & _list;
  Link * _cur;
  Link * _prev;
};

void UnlinkSeq::Unlink()
{
  assert(_cur != 0);
  if(_prev == 0)
  {
    assert(_cur == _list._pHead);
    _list._pHead = _cur->Next();
  }
  else
  {
    _prev->SetNext(_cur->Next());
  }
  delete _cur;
  _cur = 0;
}
```

9. 有助于绘制图表来查看指针在这些方法中的操纵方式。

```
void DLink::Unlink()
{
  assert(_pNext != 0);
  assert(_pPrev != 0);
  _pNext->SetPrev(_pPrev);
  _pPrev->SetNext(_pNext);
  _pPrev = this;
  _pNext = this;
}

void List::Put(int id)
{
  DLink * pLink = new DLink(id);
```

```
if(_pHead == 0)
  _pHead = pLink;
else
{
  pLink->SetNext(_pHead);
  pLink->SetPrev(_pHead->Prev());
  _pHead->Prev()->SetNext(pLink);
  _pHead->SetPrev(pLink);
  _pHead = pLink;
  }
}
int List::Get()
{
  assert(_pHead != 0);
  DLink * pLink = _pHead->Prev();
  if(pLink == _pHead)// Last one
    _pHead = 0;
  pLink->Unlink();
  int result = pLink->Id();
  delete pLink;
  return result;
}
```

10. 请注意 HSeq 构造函数中从列表数组到列表指针的隐式转换。下面是一个等价数组/指针的例子。

```
class Hseq
{
public:
  HSeq(HTable const & htab)
    : _aList(htab._aList),
      _idx(-1),
      _link(0)
  {
    NextList();
  }
  bool AtEnd()const
    { return _idx == sizeHTable && _link == 0; }
  void Advance();
  int Get(){ return _link->Id(); }
private:
  void NextList();

  List const * _aList;
  int          _idx;
  Link const * _link;
};
```

Advance 用于跟踪当前链接，但如果它到达了当前列表的结尾，它就会检索下一个非空列表。

```
void HSeq::Advance()
{
  _link = _link->Next();
  if(_link == 0)
    NextList();
}

void HSeq::NextList()
{
  assert(_link == 0);
  do
  {
    ++_idx;
  } while(_aList [_idx].IsEmpty());

  if(_idx < sizeHTable)
    _link = _aList [_idx].GetHead();
  else
    _link = 0;
}
```

11. 实现很直接。

第 4 章

1. 下面是基本类：

```
class Sieve
{
public:
  virtual int NextNumber()= 0;
};

class SourceSieve: public Sieve
{
public:
  SourceSieve(): _i(1){}
  int NextNumber();
private:
  int _i;
};

class Sieve2: public Sieve
{
```

```
public:
  Sieve2(Sieve & src): _src(src){}
  int NextNumber();
private:
  Sieve & _src;
};
```

下面是针对 SourceSieve 和 Sieve2 的 NextNumber 实现代码：

```
int SourceSieve::NextNumber()
{
  if(_i > 100)
    return -1;        // 结束
  return _i++;
}

int Sieve2::NextNumber()
{
  int i;
  do
  {
    i = _src.NextNumber();
  } while(i % 2 == 0 && i != 2 && i != -1);
  return i;
}
```

下面是 main，它打印所有小于 100 的质数：

```
int main()
{
  SourceSieve src;
  Sieve2 s2(src);
  Sieve3 s3(s2);
  Sieve5 s5(s3);
  Sieve7 s7(s5);
  int i;
  for(;;)
  {
    i = s7.NextNumber();
    if(i == -1)
      break;
    std::cout << i << " ";
  }
}
```

请访问 www.ReliSoft.com/book 获取更多的解决方案。

2. NumberCommand 的 undo 方法应该从堆栈弹出一个数。ArithmeticCommand 的 Undo 方法应该弹出结果并推回参数。注意 ArithmeticCommand 必须保存参数个数及其值。

附 B 录

事务处理

假想使用一个字处理软件的情形。你正在编辑一个大文档，已经工作了几个小时，现在想要保存编辑文档。不幸的是，当前的驱动器没有足够的空间，无法保存你的工作。你认为会发生什么情况呢？

第一个选择是：程序放弃保存并退出。实在太恐怖——数小时的工作付之东流。当你重新打开字处理软件的时候，你的心都要碎了——文档根本无法修复了。不仅丢失了当前的工作，原来的工作成果也无法挽回。

如果这样可怕的事情没有发生的话，那就是因为使用了事务处理（transaction）。事务处理包含一系列操作，它可以把程序从一种确定的状态转移到另外一种确定状态。事务处理的实现要么完全成功，要么彻底失败。如果失败了，程序必须返回到原始状态。

在所有专业字处理软件中，保存文档就是一个事务处理。如果成功保存，磁盘上的文档（文件）映像更新为文档的当前版本，而且所有内部数据结构也做相应的更新。如果保存失败，不管是什么原因，文档（文件）在磁盘上的映像保持不变，而且所有的内部数据结构也保持不变。事务处理没有成功一半的说法。如果有的话，程序和文件就不一致，就被破坏了。

我们再来考虑使用字处理软件时会碰到的另外一种情形。当你正在工作时，突然停电了，你的计算机没有 UPS（uninterruptible power supply，不间断电源），所以它也停止了工作。5 分钟后，供电恢复，计算机重新启动。你认为会发生什么呢？

最可怕的情况是，整个系统文件都被破坏了，你不得不重新格式化硬盘。当然，你的文档也就永远丢失了。不幸的是，对某些文件系统来说，这种情况是完全有可能发生。不过，大部分现代文件系统都能把破坏限制在单个目录中。如果你觉得这还不够（而且我也觉得不够），那你应当使用可恢复文件系统。这种文件系统可以把破坏限制在系统崩溃时被打开的文件中。这些文件系统通过更新文件系统元数据（如目录项）时，执行事务处理来实现这个功能。如果要求文件系统具有更强的防错功能（事务处理所有的写操作）也是不现实的——因

为这样会大大降低系统运行速度。

假设你使用的是可恢复文件系统，那么，你应当可以恢复崩溃前最后一次保存的文档版本。但如果崩溃发生在保存过程中呢？如果保存操作是作为事务处理实现的话，那么它就能保证永久数据的可靠性——可能完全是前一个保存版本，也可能是全新的保存版本。

当然，崩溃之前保存在计算机非永久性存储器中的数据是无法恢复的，它们永远丢失了（所以在字处理软件中使用自动保存功能来降低损失很重要）。不过，这并不意味程序就不能或者不应该对只与非永久性存储器有关的操作进行事务处理。实际上，每一个强健的程序为了在出现错误或者异常之后能继续运行，都必须使用事务处理。

什么样的操作需要事务处理？所有涉及到更新多数据结构的失败易发操作都需要事务处理。

暂时事务处理

暂时（或者内存之中的）事务处理不涉及程序永久状态（一般是磁盘上）的改变。因此，对系统崩溃或者掉电这样的情况来说，暂时事务处理是不可靠的。

在前面资源管理的讨论中，我们已经见过暂时事务处理。构建一个复杂数据结构的过程就是一个事务处理——要么成功，要么失败。如果任何一个子对象的构造函数失败并抛出异常，则逆向执行整个数据结构的构建，程序回到构建前状态（假设析构函数取消相应构造函数所做的一切）。这也是使用资源管理技术的另外一个好处。

不过，在某些情况下，当你要对一个操作进行事务处理时，必须执行一些特殊的操作。我们还是以字处理软件为例（顺便说一句，下面的思想同样适用于设计编辑器；什么样的程序员不会尝试编写自己的编辑器呢？）。假设文本是以段落分隔的，当用户在一个段落中间按下回车键，那么当前段落应当分成两个新段落。这个操作包括按一定顺序执行的若干步骤：

1. 分配一个新段落。
2. 分配另外一个新段落。
3. 把旧段落的第一部分拷贝到第一个新段落中。
4. 把旧段落的第二部分拷贝到第二个新段落中。
5. 把两个新段落插入旧段落的位置。
6. 删除旧段落。

切换（插入新段落）是整个操作过程中最敏感的部分。它对把所有的段落连接在一个连续文档体内的主数据结构进行操作。主数据结构很可能是一个动态数据结构，对它进行修改可能失败——当在列表中分配扩展表或者链表时，计算机的内存可能被耗尽。一旦更新了主数据结构，那么整个操作就成功了。用事务处理的术语讲就是“事务已提交”。但是如果关键更新失败，事务处理中止，程序不得不反滚（unroll）。也就是说，程序不得不取消以前所有的操作，回到原始状态（也意味着它拒绝用户的分段请求）。

设计事务处理时要确保：

- 提交之前的所有操作都能以某种安全方式取消（操作本身不必是安全的，通常也是不安全的）。
- 提交操作安全。
- 之后的所有操作也安全。

涉及内存分配的操作不安全——可能会失败，例如抛出异常。在我们的例子里，分配新段落不安全。取消操作就是删除这些段落。我们假定删除操作是安全的——它不可能失败。所以，在提交之前分配段落是可行的。

在这里，提交操作就是把新段落插入到旧段落的位置上。很可能通过一系列指针更新来实现。指针分配是一个安全操作。

提交之后的清理工作包括删除旧段落，这也是一个安全操作。注意，和往常一样，假定析构函数不会抛出异常。

实现事务处理的最好途径是创建一个事务处理对象。此对象有两种状态：提交和终止。它的初始状态通常都是中止。如果在状态变到“提交”之前，它的析构函数被调用，那么它将取消在事务处理中所执行的所有操作。显然，事务处理对象必须跟踪已经执行的所有操作——即保存动作的日志。一旦提交事务处理，此对象把它的状态变为“提交”，在此状态下，它的析构函数不会取消任何操作。

下面是分割当前段落事务处理的实现代码：

```
void Document::SplitCurPara()
{
  ParaTransaction xact;
  Paragraph * para1 = new Paragraph(_curOff);
  xact.LogFirst(para1);
  Paragraph * para2 = new Paragraph(_curPara->Size()- _curOff);
  xact.LogSecond(para2);
  Paragraph * oldPara = _curPara;
  // 可能抛出异常!
  SubstCurPara(para1, para2);
  xact.Commit();
  delete oldPara;
  // 执行 xact 析构函数
}
```

下面是事务处理对象的实现：

```
class ParaTransaction
{
public:
  ParaTransaction()
     : _commit(false), _para1(0), _para2
(0){}
  ~ParaTransaction()
```

```
{
  if(!_commit)
  {
    // 反滚所有动作
    delete _para2;
    delete _para1;
  }
}
void LogFirst(Paragraph * para){ _para1 = para; }
  void LogSecond(Paragraph * para){ _para2 = para; }
  void Commit(){ _commit = true; }
private:
  bool       _commit;
  Paragraph * _para1;
  Paragraph * _para2;
};
```

注意，我们十分小心地准备各部分事务处理。首先分配所有的资源，并把它们都记录在事务处理对象的日志上。现在事务处理拥有新段落。一旦在某处出现异常，事务处理的析构函数（仍然处于未提交状态）将取消所有的操作并释放所有的资源。

一旦我们准备好了所有资源，立即就进行切换——把新资源放置在旧资源处。切换操作一般会涉及一些指针或数组索引的操纵，但不会进行分配。一旦切换完成，我们就提交此事务处理。从这时起，事务处理不再拥有新段落。已提交事务处理的析构函数一般不会做任何事。切换之后，文档成为新段落的拥有者，同时，放弃对旧段落的拥有权，然后我们删掉旧段落。所有简单事务处理遵循下面的模式：

1. 分配所有事务处理需要的资源并记入日志（如图 B-1 所示）。
2. 把新资源切换到旧资源的位置并提交（如图 B-3 所示）。

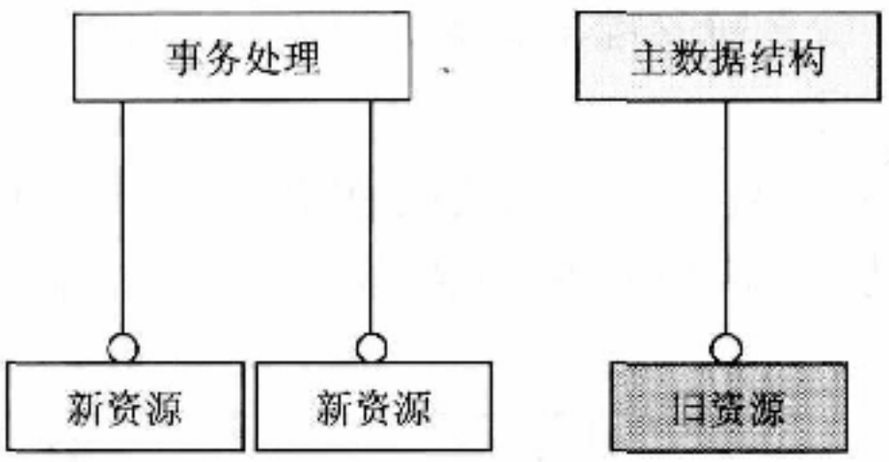


图 B-1　准备事务处理。事务处理拥有所有的新资源。主数据结构拥有旧资源

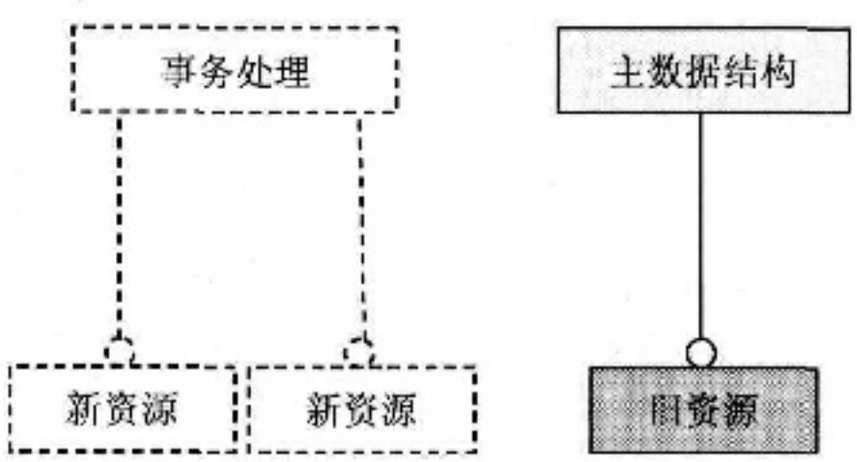


图 B-2　放弃事务处理。事务处理的析构函数释放资源

3. 清除旧资源（如图 B-4 所示）。

如果第一步不成功，事务处理终止并清除新分配资源（如图 B-2 所示）。

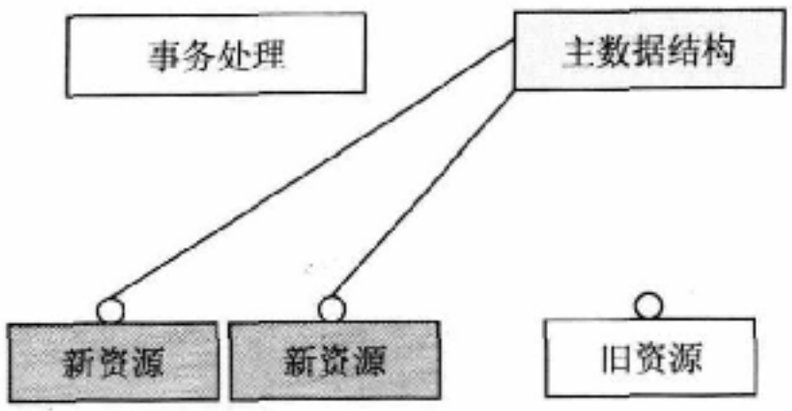


图 B-3　切换。主数据结构释放旧资源接管新资源的所有权

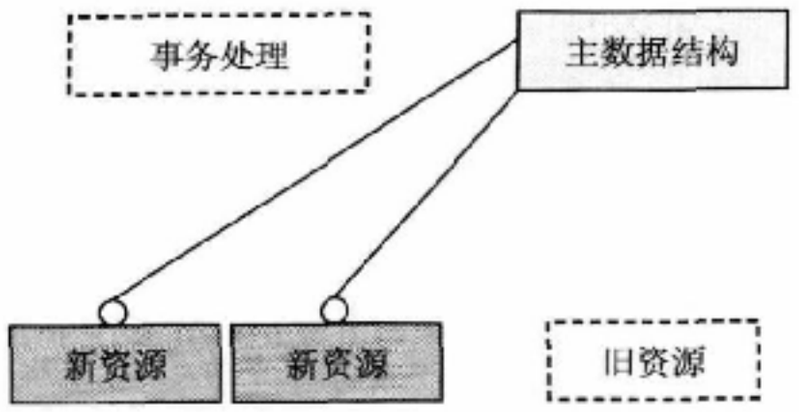


图 B-4　清理。释放旧资源、删除事务处理

永久事务处理

当设计永久事务处理（用于操纵永久数据结构）时，我们必须考虑系统崩溃或者断电等灾难的恢复问题。在这种情况下，我们并不过于关心内存中的数据（这些数据最终总会丢失），关心的是磁盘上的永久数据。

永久事务处理和暂时事务处理经历的步骤相似：

1. 准备：往磁盘写入新信息。
2. 提交：新信息变成当前信息，舍弃旧信息。
3. 清理：从磁盘删除旧信息。

系统崩溃可能发生在提交之前，也可能发生在提交之后（稍后我会解释为什么系统崩溃不会在提交期间发生）。当系统重新启动之后，我们必须找到先前中断的所有事务处理（它们肯定会在磁盘上留下一些痕迹），然后完成下面其中一个动作：如果中断是在提交事务处理之前发生的，我们就要取消所作的操作；否则，我们必须完成事务处理。这两种情况都需要清理磁盘上的一些数据。对反滚（取消操作）来说，就是删除准备事务处理而写入的信息；对完成事务处理情况来说，就是删除不再需要的旧信息。

事务处理的关键部分当然是提交。它是“切换的逆转”。在一个原子操作中，新信息成为当前信息，旧信息变为无效（如图 B-5 所示）。原子操作要么成功并在磁盘上留下永久的痕迹；要么失败，不会留下任何痕迹。你可能认为这没有什么难的，只要向文件写入某些内容，它或成功或失败，真是这样吗？

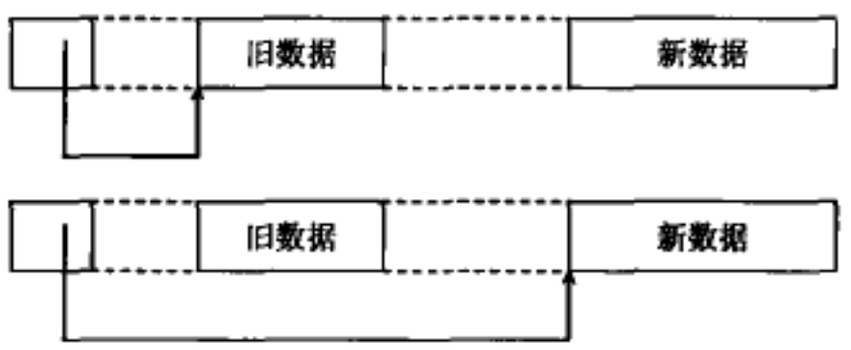


图 B-5　切换。在一个原子写操作中，磁盘上的数据结构改变了它的内容

这正是难点，事情并没有这么简单!为了更好地理解它，我们首先研究一下文件系统。首先，写文件并不意味着写入磁盘，或者至少不是立即写入磁盘。通常，写文件都是先写到缓冲器中，然后再写入高速缓存存储器中，最后才真正写到磁盘上。为了获得所使用计算机的最佳操作性能，这一切都是由运行时（在缓冲器）以及操

作系统（在高速缓存存储器）自动完成。因为与写内存相比，写磁盘的速度极其慢，所以必需高速缓存。

还有更重要的：物理磁盘写入的顺序不能确保为逻辑文件写入的顺序。实际上，文件系统根据磁头在磁盘上的物理位置安排文件写，这样避免磁头经常移来移去。文件在磁盘上的物理布局也和它邻近的逻辑形状毫无关系。更不要说写入不同的文件了，系统根本不会管程序的期望是什么，它按照任意的顺序写文件。

第三，根据文件系统设置的磁盘布局，对单个文件的连续写可以被分成好几个物理写操作。例如写入一个 32 位的数，如果它正好跨越扇区的边界，那么有可能在写磁盘的过程中，这个数只有一部分被写到磁盘上，剩下部分要等到下一次清除高速缓冲存储器时才写到磁盘上。当然，如果系统在这两次写操作中间崩溃了，你的数据就只写了一部分。好了，原子写操作就介绍这么多。

现在你已相信事务处理是不可能的，但我们可以用一些小技巧使得它最终成为可能。下面我介绍几种技巧。首先，有一种文件系统 Flush，它能百分之百地保证文件写入磁盘。虽然它不是原子写操作，但我提醒你——Flush 还是有可能在写入 32 位期间失败。不过一旦 Flush 成功了，就可以保证数据都安全地写到磁盘上了。显然，在提交事务处理之前，我们必须把新数据都“flush”到磁盘上去。否则，在文件系统崩溃之后，可能虽然提交了事务处理，但磁盘上的数据不完整。当然，必须通过另外一次“flush”来完成事务处理提交。

利用原子性情况又如何呢？有一些数据库充分利用了原子性，它们安装了自己的文件系统来支持原子写入。我们不会这样做。我们假设：如果文件足够小，那么文件写就是原子性的。“足够小”的意思是不超过一个扇区。不过为了保险起见，最好不要超过 256 个字节。这种方法在每种文件系统中都有效吗？当然不是！有些文件系统甚至还是不可恢复的。我要说的就是，这种方法在 NTFS（Windows NT 文件系统）中有效，我可以保证这一点。

下面我们将介绍最简单的永久事务处理实现方式——三文件方案。

三文件方案

一个理想的字处理软件应该读取输入文件，让用户对它进行编辑，然后保存结果。我们感兴趣的是它的保存操作。如果我们选择覆盖源文件，那么我们就会碰到麻烦。出现某类失败，得到的只是部分更新的文件（就是破坏了的文件）。

所以我们准备使用另外一个方案：把把完全更新的文档版本写入一个单独的文件中。等全部完成之后，执行 flush 操作，确保数据写入磁盘中。然后，提交事务并清理并清理原始文件。为了永久记录事务处理的状态，你还需要一个小文件。通过向此文件执行一个原子写入操作来提交事务处理。

下面就是我们的三文件方案：开始时，文件 A 包含原始数据，文件 B 没有数据，另外一个小的单字节文件 S（用于转换）所包含数据初始值为 0。开始事务处理：

1. 把文档的新版本写入文件 B。

2. Flush 文件 B，确保数据写入磁盘。

3. 提交：将“1”写入文件 S 中，并 flush 到磁盘上。

4. 清空文件 A。

文件 S 所存储数的含义是：如果为 0，则文件 A 包含有效数据；如果为 1，则文件 B 包含有效数据。当程序启动时，它根据文件 S 中的值装载相应的文件，并清空另外一个文件。整个过程就是这样。

下面我们分析一下，在此这个方案中，如果出现系统崩溃会发生什么情况。如果崩溃发生在文件 S 的新值写入磁盘之前，那么在重新启动之后，程序就会从文件 S 中读到 0。它就认为正确数据仍然位于文件 A 中，并清空文件 B。我们就回到了事务处理之前的状态。清空文件 B 就是回滚操作。

一旦 S 的新值 1 被写到磁盘上，就提交了事务处理。此时系统重新启动之后，程序从文件 S 中读到的是 1，并认为正确数据包含于文件 B 中。它将清空文件 A，于是完成了此事务处理。注意此时，文件 B 中的数据能保证是完整的，因为 S 的值为 1，表明文件 B 已经被成功地 flush 到磁盘上了。

如果我们还想继续启动一个“保存”事务处理，我们只要更换文件 A 和文件 B 的角色，并把 S 文件中的值从 1 变为 0 来提交事务处理。为了使这个方案更加可靠，切换的值不必一定是 0 和 1，可以是一些随机字节值（但必须定长）。这一技巧同样应用于布尔值的串行化中。通过这个方法，我们更容易发现磁盘数据的损坏——磁盘并不是百分之百的可靠，另外别的应用程序也有可能访问我们的文件，造成数据的损坏。冗余是防止数据损坏的第一道防线。

下面实现一个保存事务处理：

```
class SaveTrans
{
  enum State
  {
  // 一些任意位模式
      stDataInA = 0xC6,
      stDataInB = 0x3A
  };
public:
  SaveTrans()
      : _switch("Switch"), _commit(false)
  {
    _state = _switch.ReadByte();
    if(_state != stDataInA && state != stDataInB)
        throw "Switch file corrupted";
    if(_state == stDataInA)
    {
        _data.Open("A");
```

```
        _backup.Open("B");
    }
    else
    {
        _data.Open("B");
        _backup.Open("A");
    }
}
File & GetDataFile(){ return _data; }
File & GetBackupFile(){ return _backup; }
~SaveTrans()
{
  if(_commit)
      _data.Empty();
  else
      _backup.Empty();
}
void Commit()
{
  State otherState;
  if(_state == stDataInA)
      otherState = stDataInB;
  else
      otherState = stDataInA;

    _backup.Flush();
    _switch.Rewind();
    _switch.WriteByte(otherState);
    _switch.Flush();
    _commit = true;
  }
private:
  bool    _commit;
  File    _switch;
  File    _data;
  File    _backup;
  State   _state;
};
```

此处清理使用的是同一个事务处理，因为我们没有调用 Commit，这一事务处理清理正好符合我们的需要。

文件映射方案

你也许对三文件方案的性能比较关心。毕竟文档可能有几兆字节，每次保存文档时，写

入（以及 flush）磁盘都有很大的开销。所以，如果你想编写一个性能优于大部分字处理软件的程序，就必须考虑使用一种更有效的方案。

实际上，在每次保存之间，文档只改变了少数几个地方。那么，只更新这几处修改过的地方，而不重写整个文档是不是更有效呢？假设我们按“页”把文档分成若干个“块”。这里所指“页”为大小固定的 2 的某次幂。当更新一个给定块时，我们只要交换一个或者两个页。就好像浴室水池旁边的地面有稍许破损，我们只要更换破损的那几块瓷砖就行了，而不必把整个浴室地板重新铺设一遍。

严格来说，页的大小甚至不需要是 2 的固定次幂，这样可以使写磁盘 flush 效率更高，记录页更容易。每个页都用单个文件保存，但我们需要一个单独的“映射”，建立它们在文档中的顺序。所以只有“映射”能适应小切换文件，那么我们就可以通过更新映射来完成事务处理。

例如，假设一个文件有 4 页，我们想更新第 2 页。首先，我们在文件中寻找一个自由页（稍后我们会介绍事务处理如何产生自由页）。如果找不到自由页，那么我们就扩展文件添加第 5 页。然后，把更新数据写入这个自由页上。现在，我们有了文档第 2 页的两个版本：一个是在第 2 页的初始版本，另一个是在同一部分的第 5 页新版本（或者其他所使用的自由页）。然后，我们原子性地覆写映射，把第 2 页变为自由页，第 5 页取代第 2 页的位置。

如果映射不能适应小文件怎么办？没问题！我们还可以对映射应用“三文件”方案。我们可以准备映射文件的一个新版本，执行 flush 并更新交换文件来执行提交（如图 B-6 和图 B-7 所示）。

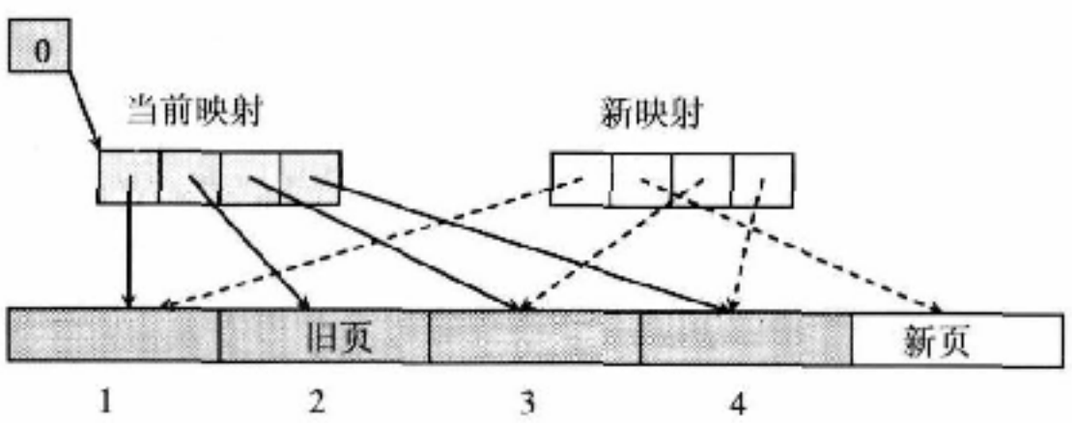


图 B-6　映射文件方案：提交前

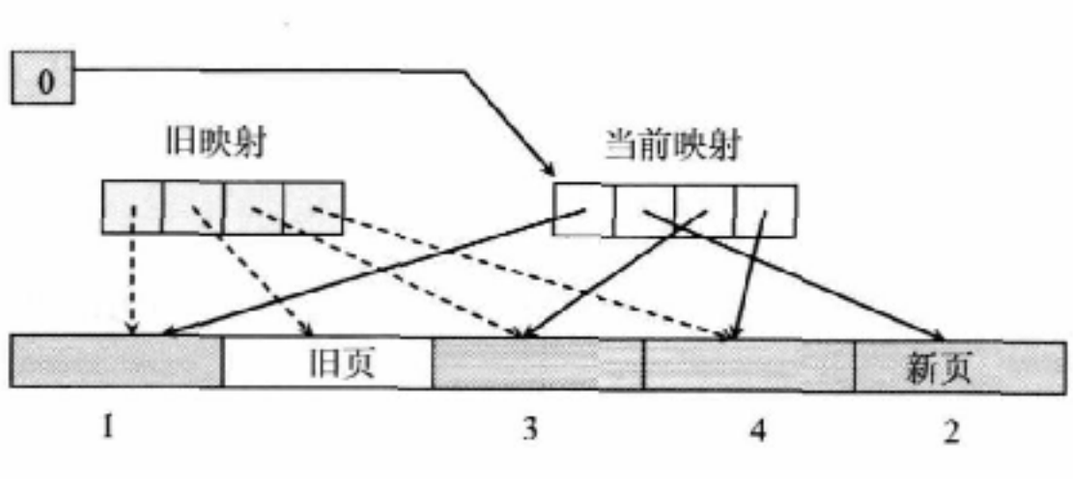


图 B-7　映射文件方案：提交后

这个方案可以扩展到多层树。事实上，很多数据库、甚至文件系统应用了相似的机制，它们都基于一种叫做二叉树（B-tree）的数据结构。